

神奇的匹配

正则表达式求精之旅

王蕾 等编著

電子工業出版社

Publishing House of Electronics Industry

北京•BEIJING

内 容 简 介

本书从正则表达式的基本概念、基本语法入手，着重于数字验证、字符串验证、数字和字符串混合验证及 HTML 处理等各个方面的应用。并基于目前流行的程序语言和应用环境（如 C#、ASP.NET、JSP、JavaScript 或 PHP），全面介绍了创建正则表达式的方法，以及正则表达式在 Web 环境中的各种应用。

本书适合广大 Web 网站开发人员、网站管理维护人员和在校学生阅读，尤其适合与字符串处理相关的 Web 编程技术人员阅读。

未经许可，不得以任何方式复制或抄袭本书之部分或全部内容。
版权所有，侵权必究。

图书在版编目（CIP）数据

神奇的匹配：正则表达式求精之旅 / 王蕾等编著. —北京：电子工业出版社，2014.8
ISBN 978-7-121-23656-3

I. ①神… II. ①王… III. ①正则表达式—研究 IV. ①TP301.2

中国版本图书馆 CIP 数据核字（2014）第 139900 号

策划编辑：张月萍

责任编辑：徐津平

印 刷：三河市双峰印刷装订有限公司

装 订：三河市双峰印刷装订有限公司

出版发行：电子工业出版社

北京市海淀区万寿路 173 信箱

邮编：100036

开 本：787×1092 1/16

印张：21.5

字数：550 千字

版 次：2014 年 8 月第 1 版

印 次：2014 年 8 月第 1 次印刷

印 数：3500 册

定价：59.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888。

质量投诉请发邮件至 zltz@phei.com.cn，盗版侵权举报请发邮件到 dbqq@phei.com.cn。

服务热线：（010）88258888。

前言

正则表达式（正规表示法、常规表示法），在代码中通常简写为 `regex`、`regexp` 或 `RE`，它使用单个字符串来描述、匹配一系列符合某个句法规则的字符串。在很多文本编辑器中，正则表达式通常被用来检索、替换那些符合某个模式的文本，并且，正则表达式已经得到许多脚本语言、编程语言和数据库的良好支持。

正则表达式在过去十多年间越来越普及，如今所有常用的编程语言都会包含一个强大的正则表达式函数库，有的甚至其语言本身就内嵌了对正则表达式的支持。许多开发人员都会利用这些正则表达式的功能，在应用程序中为用户提供使用正则表达式对其数据进行查找或者过滤的功能。随着正则表达式的广泛应用，市面上出现了许多相关的著作，大多数这类书籍都能很好地讲解正则表达式的语法，并且还会提供一些示例以及参考文献。然而，还没有任何一本书能够面向现实世界中使用的计算机，以及在各种互联网应用中遇到的实际问题，为读者提供基于正则表达式的解决方案。

本书详细解释了正则表达式的各个组成部分、含义、如何使用它们，以及在编写正则表达式时如何避免常见的错误，并列举了丰富的示例，打破了正则表达式难以掌握的传统神话。读者将学习到如何有效地驾驭正则表达式所提供的强大功能，并且全面理解正则表达式的高灵活性和无限潜能。

正则表达式的特点

- （1）灵活性、逻辑性和功能性非常强。
- （2）可以迅速地用极简单的方式控制复杂的字符串。
- （3）对于刚接触的人来说，比较晦涩难懂。

本书的特点

1. 覆盖内容多

覆盖了正则表达式的全部基本理论和语法，并给出了不同语言环境（如 `C#`、`Java`、`JavaScript` 或 `PHP` 等）下的各自相应的基本理论和语法。

2. 多环境全面应用

几乎覆盖了正则表达式的所有应用，如数字处理、字符串处理、数字和字符串混合处理、`HTML` 处理和编码处理等。同时，本书讲解了在不同环境下（如 `ASP.NET`、`JSP`、`PHP` 和客户端）正则表达式的应用。

3. 示例丰富

书中提供了大量正则表达式示例，数量多达 540 个。用户可以直接从书中找到所需的正则表

达式。

4. 讲解详细、逻辑严整

本书详细分析和讲解了书中每一个正则表达式示例，并使用正则表达式测试工具进行了测试。

5. 实用性强

书中的每一个正则表达式示例都是基于当前的流行应用，读者不需要任何修改就可以直接使用。

本书的内容

第1章 介绍了正则表达式的基本概念、测试正则表达式的工具及基础理论，如元字符、字符类、字符转义、反义、限定符、替换、分组、反向引用、零宽度断言、负向零宽度断言、匹配选项、注释、优先级顺序、递归匹配等。这些理论将为编写正则表达式提供法则和规范。

第2章 介绍了数字验证，以及与数字相关的字符串验证，如数值验证、电话号码验证、邮政编码验证、IP 地址验证、身份证号码验证和银行卡号验证。

第3章 介绍了字符串的验证，如英文单词的验证、文件名称的验证、网络常用元素的验证、非单词字符串的验证等。被验证的字符串可能包含英文单词字符、数字字符和特殊字符。特殊字符是指除英文单词字符和数字字符之外的字符，如/、\、|、,和:等。

第4章 介绍了由数字和非数字字符组成的字符串的混合验证，如日期和时间验证、车牌号码验证、数学表达式验证和编码规范验证等。

第5章 介绍了与 HTML 元素相关的验证和处理。首先介绍了验证 HTML 元素的基础知识，如 HTML 标记验证、非封闭 HTML 标记验证、封闭 HTML 标记验证、属性赋值表达式验证、注释表达式验证和脚本代码验证等。其次介绍了部分 HTML 元素的验证，如
、<hr>、<a>和<input>等元素的验证。最后介绍处理 HTML 元素的方法，如提取网页中的标题、链接、图片等。

第6章 介绍了 C#常用数据类型检查与转换，如数值数据类型检查与转换、布尔数据类型检查与转换、时间数据类型检查与转换等。

第7章 介绍了不变字符串（由 String 类表示）和可变字符串（由 StringBuilder 类表示）的处理方法。不变字符串对象一旦被创建，那么该对象是不能被修改的；而可变字符串对象被创建之后，开发人员可以对该对象进行修改，如追加、移除、替换和插入等操作。

第8章 介绍了使用正则表达式类 Regex 来验证、匹配、替换、分组和分割给定字符串的方法，以及使用正则表达式验证给定的字符串是否满足给定要求。最后，还介绍了使用正则表达式从网页中提取网页标题、HTTP 地址、图像地址等信息的方法。

第9章 介绍了 ASP.NET 中的验证方法和服务器端控件。ASP.NET 提供了一组验证控件，并提供了一种易用但功能强大的检错方式，同时在必要时还可以向用户显示错误信息。ASP.NET 验证提供了非空验证、范围验证、比较验证、正则表达式验证、用户自定义验证等多种验证方式，以及用于处理或集中显示页面上所有验证控件的提示或错误信息功能。

第10章 介绍了 PHP 的数据类型。同时，还讲解了 PHP 中各种数据的转换。

第11章 介绍了 PHP 中字符串的各种处理。通过这些处理，可以实现一些简单的字符串检验功能。

第 12 章 介绍了在 PHP 中如何使用正则表达式，内容包括 POSIX 库和 PCRE 库的使用。

第 13 章 介绍了在 JSP 中，Java 语言的数据类型、数据类型转换和字符串的操作。

第 14 章 介绍了在 JSP 中，如何使用 Java 语言进行正则表达式验证。其中，详细讲解了 Pattern 类和 Matcher 类的使用，以及 JSP 中正则表达式的应用举例。

第 15 章 介绍了客户端脚本 JavaScript 的数据类型和数据类型的转换。

第 16 章 介绍了 JavaScript 中常见的字符串操作和数据处理。

第 17 章 介绍了常见的 JavaScript 正则表达式的应用。

适合的读者

- 从事字符串处理、开发或研究的相关人员
- 网页设计和制作人员
- 网页制作爱好者
- 网站维护人员
- 网站建设及网络开发人员
- 进行毕业设计的学生

本书由王蕾（东华理工大学）编写，同时参与编写的还有张燕、杜海梅、孟春燕、吴金艳、鲍凯、庞雁豪、杨锐丽、鲍洁、王小龙、李亚杰、张彦梅、刘媛媛、李亚伟和张昆，在此一并表示感谢。

目 录

第 1 篇 正则表达式基础

第 1 章 外行看正则表达式	1
1.1 什么是正则表达式	1
1.2 本书使用的测试工具	2
1.3 理解元字符	3
1.3.1 匹配位置的元字符	3
1.3.2 匹配字符的元字符	4
1.3.3 元字符总结	5
1.4 文字匹配	7
1.4.1 字符类	7
1.4.2 字符转义	9
1.4.3 反义	9
1.4.4 限定符	10
1.5 字符的操作	15
1.5.1 替换	15
1.5.2 分组	17
1.5.3 反向引用	17
1.6 正则的其他操作	19
1.6.1 零宽度断言	19
1.6.2 负向零宽度断言	20
1.6.3 匹配选项	21
1.6.4 注释	21
1.6.5 优先级顺序	22
1.6.6 递归匹配	22
1.7 典型正则表达式解释	23
1.7.1 匹配 Windows 操作系统的名称	23
1.7.2 匹配 HTML 标记	23
1.7.3 匹配 HTML 标记之间的内容	24
1.7.4 匹配 CSV 文件内容	25
第 2 章 数字验证的方法	27
2.1 9 种数值验证	27

2.1.1	字符串只包含数字的验证	27
2.1.2	字符串只包含整数的验证	31
2.1.3	字符串指定范围的整数验证	32
2.1.4	字符串为实数的验证	35
2.1.5	字符串指定精度的实数验证	38
2.1.6	科学计数法的数值验证	39
2.1.7	二进制数值的验证	39
2.1.8	八进制数值的验证	40
2.1.9	十六进制数值的验证	41
2.2	4 种国内电话号码的验证	43
2.2.1	国内手机号码验证	43
2.2.2	固定电话号码（不包括区号）验证	44
2.2.3	区号+固定电话号码验证	45
2.2.4	区号+固定电话号码+分机号码验证	47
2.2.5	固定电话号码验证总结	48
2.3	2 种身份证号码验证	49
2.3.1	15 位身份证号码验证	50
2.3.2	18 位身份证号码验证	51
2.4	银行卡和信用卡号验证	51
2.5	邮政编码验证	52
2.5.1	国内邮政编码验证	52
2.5.2	国际邮政编码验证	53
2.6	4 种 IP 地址验证	53
2.6.1	简单 IP 地址验证	53
2.6.2	精确 IP 地址验证	54
2.6.3	子网内部 IP 地址验证	56
2.6.4	64 位 IP 地址验证	57
第 3 章	常用字符串验证方法	60
3.1	5 种英文单词验证	60
3.1.1	小写英文单词验证	60
3.1.2	大写英文单词验证	61
3.1.3	英文单词的分隔符验证	62
3.1.4	否定验证	64
3.1.5	具有重复特征的英文单词验证	67
3.2	6 种非单词字符串验证	69
3.2.1	英文标点符号验证	69
3.2.2	中文标点符号验证	70
3.2.3	中文文本验证	70
3.2.4	特殊字符验证	71

3.2.5	密码验证	71
3.2.6	字符表的分类	76
3.3	常用的文件名称和路径验证	76
3.3.1	通配符	76
3.3.2	指定文件扩展名的验证	77
3.3.3	指定文件名的验证	78
3.3.4	包含指定字符串的文件全名验证	78
3.3.5	排除两端存在空白字符的文件全名验证	79
3.3.6	文件路径验证	81
3.4	4 种网络常用元素验证	82
3.4.1	电子邮件验证	82
3.4.2	主机名称验证	83
3.4.3	HTTP 地址验证	85
3.4.4	FTP 地址验证	86
第 4 章	常见数字和字符混合验证	87
4.1	5 种数学表达式验证	87
4.1.1	操作数验证	87
4.1.2	操作符验证	88
4.1.3	简单数学表达式验证	88
4.1.4	只含操作数和操作符的数学表达式验证	89
4.1.5	包含小括号的数学表达式验证	90
4.2	8 种日期和时间验证	92
4.2.1	年验证	92
4.2.2	月验证	93
4.2.3	日验证	94
4.2.4	年月日格式的日期验证	95
4.2.5	24 小时制时分秒格式的时间验证	96
4.2.6	12 小时制时分秒格式的时间验证	97
4.2.7	带毫秒的时间验证	98
4.2.8	长格式的日期和时间验证	100
4.3	4 种编码规范验证	101
4.3.1	类名称验证	101
4.3.2	声明变量表达式验证	102
4.3.3	函数名称验证	102
4.3.4	声明函数表达式验证	103
4.4	3 种车牌号码验证	106
4.4.1	通用车牌号码验证	106
4.4.2	武警车牌号码验证	108

第 5 章 常见的 HTML 元素验证和处理	110
5.1 6 种 HTML 元素验证的基础	110
5.1.1 HTML 标记验证	110
5.1.2 非封闭 HTML 标记验证	111
5.1.3 封闭 HTML 标记验证	112
5.1.4 属性赋值表达式验证	113
5.1.5 HTML 中的注释验证	116
5.1.6 HTML 中的脚本代码块验证	117
5.2 4 种非封闭的 HTML 元素验证	118
5.2.1 元素验证	119
5.2.2 <hr>元素验证	121
5.2.3 <a>元素验证	124
5.2.4 <input>元素验证	128
5.3 封闭的 HTML 元素验证	129
5.4 处理 HTML 元素	130
5.4.1 提取 HTML 标记	130
5.4.2 提取 HTML 标记之间的内容	131
5.4.3 提取 URL	132
5.4.4 提取图像的 URL	133
5.4.5 提取 HTML 页面的标题	134

第 2 篇 ASP.NET 正则表达式应用

第 6 章 C#常用数据类型的检查与转换	136
6.1 数值数据类型的检查与转换	136
6.1.1 整数检查	136
6.1.2 实数检查	137
6.1.3 整数和字符串之间的转换	138
6.1.4 浮点数和字符串之间的转换	140
6.2 布尔数据类型检查与转换	141
6.2.1 布尔值检查	141
6.2.2 布尔值和字符串之间的转换	141
6.3 时间数据类型检查与转换	142
6.3.1 时间数据类型检查	142
6.3.2 时间和字符串之间的转换	143
6.4 数据类型检查与转换应用实例	144
第 7 章 不可变字符串与可变字符串的处理	146
7.1 15 种不可变字符串 String 处理	146

7.1.1	String 类和对象	146
7.1.2	插入字符串	147
7.1.3	替换字符串	147
7.1.4	填充字符串	148
7.1.5	删除字符串	149
7.1.6	分割字符串	149
7.1.7	比较字符串	150
7.1.8	连接字符串	151
7.1.9	处理字符串中的空白	152
7.1.10	转换字符串大小写	153
7.1.11	匹配和检索字符串	153
7.1.12	格式化字符串	156
7.1.13	获取子字符串	156
7.1.14	编码字符串	157
7.1.15	不可变字符串 String 处理的应用	157
7.2	8 种可变字符串 StringBuilder 处理	159
7.2.1	StringBuilder 类和对象	159
7.2.2	追加字符串	159
7.2.3	插入字符串	161
7.2.4	替换字符串	162
7.2.5	删除字符串	163
7.2.6	复制字符串	164
7.2.7	处理字符串容量	165
7.2.8	可变字符串 StringBuilder 处理的应用	165
第 8 章	常见的.NET 框架中正则表达式及其应用	167
8.1	10 种.NET 框架中的正则表达式类库	167
8.1.1	System.Text.RegularExpressions 命名空间	167
8.1.2	正则表达式类 Regex	168
8.1.3	正则表达式选项	168
8.1.4	检查是否匹配表达式	169
8.1.5	匹配单个匹配项	170
8.1.6	匹配多个匹配项	171
8.1.7	替换	173
8.1.8	使用委托 MatchEvaluator 处理匹配结果	174
8.1.9	获取分组名称	175
8.1.10	分割表达式	175
8.2	14 种正则表达式类 Regex 处理字符串	176
8.2.1	只包含数字验证	176
8.2.2	整数验证	176

8.2.3	实数验证	176
8.2.4	电话号码验证	177
8.2.5	邮政编码验证	177
8.2.6	身份证号码验证	177
8.2.7	银行卡号验证	177
8.2.8	IP 地址验证	178
8.2.9	日期和时间验证	178
8.2.10	车牌号码验证	178
8.2.11	电子邮件验证	179
8.2.12	URL 验证	179
8.2.13	提取网页标题	179
8.2.14	提取网页中的图像地址	180
8.2.15	提取网页中的 HTTP 地址	181
第 9 章	常见 ASP.NET 验证控件	183
9.1	ASP.NET 验证简介	183
9.2	2 种非空验证	184
9.2.1	无初始值的非空验证	185
9.2.2	指定初始值的验证	185
9.3	3 种范围验证	186
9.3.1	整数范围验证	186
9.3.2	字母范围验证	187
9.3.3	日期范围验证	188
9.4	3 种比较验证	189
9.4.1	两个控件内容的比较验证	189
9.4.2	检查控件内容的数据类型	190
9.4.3	指定的值和控件内容的比较验证	191
9.5	2 种自定义验证	192
9.5.1	自定义客户端验证	192
9.5.2	自定义服务端验证	193
9.6	7 种正则表达式验证	194
9.6.1	整数验证	194
9.6.2	数值验证	195
9.6.3	电话号码验证	196
9.6.4	身份证号码验证	197
9.6.5	电子邮件验证	198
9.6.6	日期和时间验证	199
9.6.7	URL 验证	200
9.7	2 种显示验证摘要	201
9.7.1	在对话框上显示验证摘要	201

9.7.2 在网页上显示验证摘要	202
------------------------	-----

第 3 篇 PHP 正则表达式应用

第 10 章 常见 PHP 数据类型	204
10.1 7 种 PHP 常见数据类型	204
10.1.1 布尔型	204
10.1.2 NULL 型	204
10.1.3 整型	205
10.1.4 浮点型	205
10.1.5 字符串	205
10.1.6 数组	207
10.1.7 对象	208
10.2 5 种常见的类型转化	209
10.2.1 变量类型变化	209
10.2.2 强制类型转换	210
10.2.3 字符串转化	212
10.2.4 数字转化	212
10.2.5 数组转化	213
10.3 小结	213
第 11 章 常见 PHP 字符串处理	214
11.1 常见的 3 种字符串分析	214
11.1.1 访问字符串中的字符	214
11.1.2 处理子字符串	215
11.1.3 分割字符串	217
11.2 4 种字符串的操作	218
11.2.1 删除字符串的空白	218
11.2.2 转换字符串大小写	219
11.2.3 填补字符串	220
11.2.4 反转字符串	221
11.3 2 种字符串的格式化	221
11.3.1 格式化数字	221
11.3.2 格式化字符串	222
11.4 字符串的查找和替换	224
11.4.1 查找字符串	224
11.4.2 替换字符串	225
11.5 3 种常见的字符串的比较方法	227
11.5.1 按 ASCII 码顺序比较	227

11.5.2	按“自然排序”法比较	228
11.5.3	按相似性比较	229
11.6	处理 HTML 和 URL	230
11.6.1	HTML 标签的清理	230
11.6.2	HTML 实体的处理	231
11.6.3	URL 字符串的解析	232
11.6.4	URL 编码处理	234
11.6.5	查询字符串的构造	235
11.7	小结	236
第 12 章	PHP 与正则表达式的应用	237
12.1	关于 POSIX 扩展库的正则表达式函数	237
12.1.1	模式匹配	237
12.1.2	模式替换	238
12.1.3	模式分割	239
12.2	关于 PCRE 库的正则表达式函数	239
12.2.1	对正则表达式匹配	240
12.2.2	取得正则表达式的全部匹配	241
12.2.3	返回与模式匹配的数组单元	241
12.2.4	正则表达式的替换	242
12.2.5	正则表达式的拆分	243
12.3	PHP 与正则表达式的综合应用	243
12.3.1	表单验证	243
12.3.2	UBB 代码	247
12.3.3	分析 Apache 日志文件	251
12.4	小结	254

第 4 篇 JSP 正则表达式应用

第 13 章	常见的 JSP 中数据处理	255
13.1	5 种 JSP 中的常用数据类型	255
13.1.1	整数类型及应用	255
13.1.2	浮点类型及应用	258
13.1.3	字符类型及应用	259
13.1.4	布尔类型及应用	260
13.1.5	字符串类型及应用	261
13.2	2 种 JSP 中数据类型的转换	262
13.2.1	自动类型转换及应用	262
13.2.2	强制类型转换及应用	263

13.3	7 种 JSP 中字符串数据的处理	264
13.3.1	字符串与其他类型数据的转换	264
13.3.2	字符串的分析	265
13.3.3	字符串的查找与替换	267
13.3.4	字符串数据的整理	268
13.3.5	字符串的比较	269
13.3.6	字符串的连接	270
13.3.7	字符串的格式化	271
13.4	小结	272
第 14 章	常见的 JSP 中正则表达式	273
14.1	2 种 JSP 中的正则表达式函数	273
14.1.1	Pattern 类	273
14.1.2	Matcher 类	275
14.1.3	正则表达式常用的四种功能	278
14.2	JSP 中正则表达式的常见应用示例	282
14.2.1	电子邮件地址的校验	282
14.2.2	URL 地址的校验	283
14.2.3	电话号码的校验	284
14.3	小结	286
第 5 篇 JavaScript 正则表达式应用		
第 15 章	常见的 JavaScript 中数据类型及其转化	287
15.1	常见的三种 JavaScript 数据类型	287
15.1.1	数字基本类型	287
15.1.2	字符串基本类型	288
15.1.3	布尔值基本类型	289
15.2	数据类型转化	289
15.2.1	基本数据类型转换	289
15.2.2	将字符串转化为整数	290
15.2.3	将字符串转化为浮点数	290
第 16 章	常见 JavaScript 字符串和数组处理	291
16.1	6 种字符串格式处理	291
16.1.1	获取字符串的长度	291
16.1.2	根据指定的 Unicode 编码返回一个字符串	291
16.1.3	将字符串分割并存储到数组中	292
16.1.4	比较两个字符串的大小	292

16.1.5	将字符串转化为小写格式	293
16.1.6	将字符串转化为大写格式	294
16.2	最基本的字符串查找、替换	294
16.2.1	获取指定字符（串）第一次在字符串中出现的位置	294
16.2.2	获取指定字符（串）最后一次在字符串中出现的位置	295
16.2.3	替换字符串中指定的内容	296
16.3	字符串截取、组合的方法	296
16.3.1	返回字符串中指定位置处的字符	297
16.3.2	将一个或多个字符串连接到当前字符串的末尾	298
16.3.3	获取指定位置的字符的 Unicode 编码	298
16.3.4	从字符串中提取子串（1）	299
16.3.5	从字符串中提取子串（2）	300
16.3.6	从字符串中提取子串（3）	301
16.4	字符串 HTML 格式化	301
16.4.1	在字符串两端加入锚点标志	302
16.4.2	在字符串的两端加上粗体标志	302
16.4.3	在字符串两端加入斜体标签	302
16.4.4	在指定字符串的两端加上大字体标志	303
16.4.5	在字符串的两端加上固定宽度字体标记	303
16.4.6	设置字符串输出时的字体大小	304
16.4.7	设置字符串输出时的前景色	305
16.4.8	在字符串上加入超链接	305
16.4.9	在字符串两端加上小字体标记	306
16.4.10	在字符串两端加入下标标签	307
16.4.11	在字符串两端加入上标标签	307
16.4.12	在字符串的两端加入下划线标记	308
16.5	Array 对象的方法及使用	308
16.5.1	连接其他数组到当前数组末尾	309
16.5.2	将数组元素连接为字符串	309
16.5.3	删除数组中的第一个元素	310
16.5.4	删除数组中最后一个元素	310
16.5.5	删除或替换数组中部分数据	311
16.5.6	将指定的数据添加到数组中	312
16.5.7	在数组前面插入数据	313
16.5.8	获取数组中的一部分数据	313
16.5.9	反序排列数组中的元素	314
16.5.10	对数组中的元素进行排序	314
16.5.11	返回一个包含数组中全部数据的字符串	315

第 17 章 常见 JavaScript 正则表达式应用	317
17.1 正则表达式对象 RegExp 及其应用	317
17.1.1 正则表达式的创建	317
17.1.2 判断字符串中是否存在匹配内容	317
17.1.3 对字符串进行匹配检测	318
17.1.4 编译正则表达式	319
17.1.5 替换字符串中的指定内容	320
17.2 处理匹配的结果	320
17.2.1 获取字符串中所有的匹配信息	320
17.2.2 获取第一次匹配的起始位置 (1)	321
17.2.3 获取第一次匹配的起始位置 (2)	322
17.2.4 获取子匹配的结果	322
17.2.5 获取与正则表达式进行匹配检测的字符串	324
17.2.6 获取最近一次匹配的内容	324
17.2.7 获取最近一次匹配的最后一个子匹配	325
17.2.8 获取匹配的内容的最后一个索引位置	325
17.2.9 获取匹配内容左侧的字符信息	326
17.2.10 获取匹配内容右侧的字符信息	327

第 1 篇 正则表达式基础

第 1 章 外行看正则表达式

正则表达式在程序设计语言中有着广泛的应用，通常用来处理字符串，如匹配字符串、查找字符串、替换字符串等。可以说，正则表达式是一段文本或一个公式，它是用来描述用某种模式去匹配一类字符串的公式，并且该公式具有一定的模式。本章讲解正则表达式的基本规则。

1.1 什么是正则表达式

正则表达式 (Regular Expression) 起源于人类神经系统的早期研究。神经生理学家 Warren McCulloch 和 Walter Pitts 研究出一种使用数学方式描述神经网络的方法。1956 年，数学家 Stephen Kleene 发表了一篇标题为“神经网络事件的表示法”的论文，并在该论文中引入了“正则表达式”这一概念。该论文称正则表达式是：“正则集的代数”的表达式，因此，采用“正则表达式”这个术语。正则表达式的定义存在多种说法，具体如下。

- ❑ 正则表达式就是用某种模式去匹配一类字符串的公式，主要用来描述字符串匹配的工具。
- ❑ 正则表达式描述了一种字符串匹配的模式。它可以用来检查字符串中是否含有某种子串、将匹配的子串做替换或者从某个字符串中取出符合某个条件的子串等。
- ❑ 正则表达式是由普通字符（如字符 a~z）和特殊字符（称为元字符）组成的文字模式。正则表达式作为一个模板，将某个字符模式与所搜索的字符串进行匹配。
- ❑ 正则表达式就是用于描述某些规则的工具，这些规则通常用于处理字符串中的查找或替换字符串。换句话说，正则表达式就是记录文本规则的代码。
- ❑ 正则表达式就是用一个“字符串”来描述一个特征，然后去验证另一个“字符串”是否符合这个特征。

学过《编译原理》的读者可能知道不确定有限自动机 (Non-deterministic finite automaton, 简称 NFA) 和确定有限自动机 (Deterministic finite automaton, 简称 DFA)。其实，正则表达式是一个不确定有限自动机。NFA 和 DFA 的最大区别在于它们的状态转换函数。NFA 可以对同一个字符串产生多种理解方式，而 DFA 则只有唯一的一种理解方式。也正因为如此，NFA 在匹配过程中可能会回溯，NFA 的效率一般要低于 DFA。因此，在书写正则表达式时应尽量减少回溯来提高正则表达式的效率。

如果你在 Windows 或 DOS 下使用过用于文件查找的通配符*或?，那么就不难理解正则表达式。如果需要查找所有 Word 文档，那么你可能使用表达式*.doc，其中，字符*是一个通配符，它可以代表任意字符串。正则表达式和通配符具有相似性，它可以使用一些字符（如字符.）表示任意字符，并且，它比通配符更具有精确性。

在正则表达式中，匹配是最常用的一个词语，它描述了正则表达式的动作结果。给定一段文

本或字符串，使用正则表达式从文本或字符串中查找出符合正则表达式的字符串。有可能文本或字符存在不止一个部分满足给定的正则表达式，这时每一个这样的部分都被称为一个匹配。匹配分为以下三种类型。

- ❑ 形容词性的匹配，即一个字符串匹配一个正则表达式。
- ❑ 动词性的匹配，即在文本或字符串里匹配正则表达式。
- ❑ 名词性的匹配，即字符串中满足给定的正则表达式的一部分。

正则表达式的应用非常广泛，特别是在字符串处理方面。目前，正则表达式在很多软件中都已得到广泛应用，如 Linux、UNIX、HP 等运算系统，C#、PHP、Java 等程序开发环境，以及在很多的应用软件中，都可以看到正则表达式的不同应用。正则表达式常见的应用如下。

- ❑ 验证字符串，即验证给定的字符串或子字符串是否符合指定特征，例如，验证是否是合法的邮件地址、验证是否为合法的 HTTP 地址等。
- ❑ 查找字符串，从给定的文本中查找符合指定特征的字符串，比查找固定字符串更加灵活方便。
- ❑ 替换字符串，即把给定字符串中的符合指定特征的字字符串替换为其他字符串，比普通的替换更强大。
- ❑ 提取字符串，即从给定的字符串中提取符合指定特征的字字符串。

1.2 本书使用的测试工具

创建一个正则表达式之后，需要测试该正则表达式是否正确。本书中，使用正则表达式测试工具“Code Architects Regex Tester”来测试正则表达式。这是一个专门用来测试正则表达式的工具，它的初始界面如图 1.1 所示。此时，该工具包含 4 个区域：**Regex**、**Replace**、**Source** 和 **Matches**。其中，**Regex** 区域用来书写正则表达式，**Replace** 区域用来书写替换相关的表达式，**Source** 区域用来输入被测试的字符串或文本，**Matches** 区域则显示测试或匹配的结果。

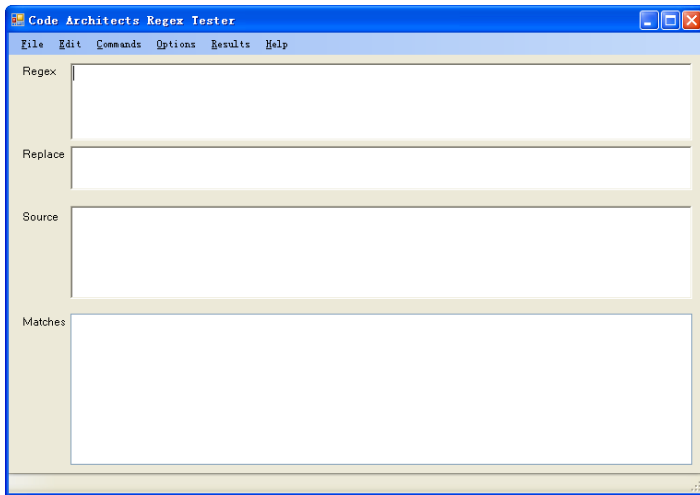


图 1.1 Code Architects Regex Tester 工具初始界面

单击“Results”→“Auto”命令，运算如图 1.2 所示。Code Architects Regex Tester 工具只显示了三个区域：**Regex**、**Source** 和 **Matches**，如图 1.3 所示。

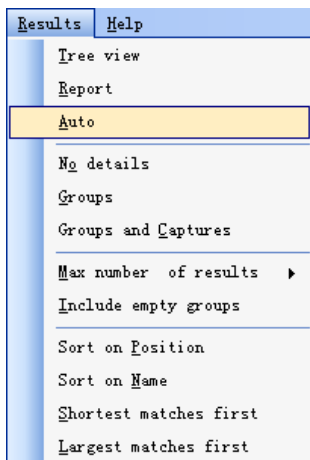
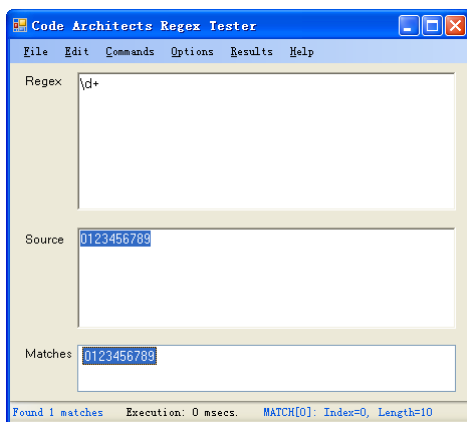


图 1.2 单击“Auto”命令



图 1.3 Code Architects Regex Tester 工具只显示 3 个区域的界面

使用 Code Architects Regex Tester 工具测试正则表达式 `\d+`。其中，被测试的字符串为“0123456789”。测试结果如图 1.4 所示。

图 1.4 测试正则表达式“`\d+`”

1.3 理解元字符

在正则表达式中，元字符（Metacharacter）是一类非常特殊的字符，它能够匹配一个位置或字符集合中的一个字符，如 `.`、`\w` 等。根据功能，元字符可分为两种类型：匹配位置的元字符和匹配字符的元字符。

1.3.1 匹配位置的元字符

匹配位置的元字符包括 3 个字符：`^`、`$` 和 `\b`。其中，`^`（脱字符号，通常在文章中插入字时使用）和 `$`（美元符号）只匹配一个位置，它们分别匹配行的开始和结尾。

以下正则表达式匹配以“String”开始的行，即被匹配的行的第一个字符串为“String”。

```
^String (1)
```

以下正则表达式匹配以“String”结尾的行，即被匹配的行的最后一个字符串为“String”。

```
String$ (2)
```

以下正则表达式匹配以“String”开始和结尾的行，即被匹配的行的第一个字符串和最后一个字符串都为“String”。实际上，该行只包含字符串“String”。

```
^String$ (3)
```

以下正则表达式匹配一个空行，该行中不包含任何字符串。

```
^$ (4)
```

以下正则表达式匹配任意行。该表达式只匹配行中的开始位置，因为任意行都包括其开始位置，所以该表达式将匹配任意行。

```
^ (5)
```

元字符**b**和[^]、^{\$}具有相似性，它也是匹配一个位置。**b**可以匹配单词的开始或结尾，即单词的分界处。通常情况下，英文单词之间往往由空格符号、标点符号或换行符号来分隔，但是元字符**b**不匹配空格符号、标点符号和换行符号中的任何一个，它仅仅匹配一个位置。

以下正则表达式匹配以“Str”开头的字符串，如“String”、“String Format”等。

```
\bStr (6)
```

正则表达式**bStr**匹配的字符串必须以“Str”开头，并且“Str”之前是单词的分界处。正则表达式**bStr**不能描述或限定“Str”之后的字符串的形式。以下正则表达式匹配以“ing”结尾的字符串，如“String”、“This is a String”等。

```
ing\b (7)
```

正则表达式 **ing\b** 匹配的字符串必须以“ing”结尾，且“ing”之后是单词的分界处。以下正则表达式匹配一个完整的单词“String”。

```
\bString\b (8)
```

注意：在某些特定环境或语言下，还可以分别采用[<]和[>]来匹配单词的开始位置和结束位置。它们在效果上和元字符**b**等效，即都匹配单词的边界的两个位置，即开始位置和结束位置。

1.3.2 匹配字符的元字符

匹配字符的元字符包括 7 个字符：[.]（点号）、^{\w}、^{\W}、^{\s}、^{\S}、^{\d}和^{\D}。其中，[.]（点号）匹配除换行符之外的任意字符；^{\w}匹配单词字符（包括字母、数字、下画线和汉字）；^{\W}匹配任意的非单词字符；^{\s}匹配任意的空白字符，如空格、制表符、换行符、中文全角空格等；^{\S}匹配任意的非空白字符；^{\d}匹配任意的数字；^{\D}匹配任意的非数字字符。

以下正则表达式匹配一个非空行，该行中可以包含除换行符之外的任意字符。

```
^. $ (9)
```

以下正则表达式匹配一个非空行，且该行中只能包含字母、数字、下画线和汉字中的任意字符。

```
^\w$ (10)
```

以下正则表达式匹配以字母“a”开头的长度等于 8 的任意单词。

```
\ba\w\w\w\w\w\w\w\b (11)
```

正则表达式**ba\w\w\w\w\w\w\w\b**匹配单词“anterior”的方式如图 1.5 所示。

以下正则表达式匹配以字母“a”开头、后跟随形如“3 个字符”+“3 个字符”+“1 个非数字字符”、长度等于 8 的任意单词。

```
\ba\w\w\w\d\d\d\D\b (12)
```

正则表达式**ba\w\w\w\d\d\d\D\b**匹配字符串“ante123_”的方式如图 1.6 所示。

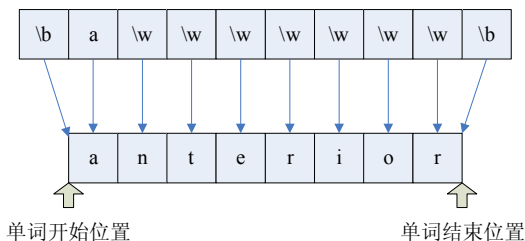


图 1.5 \ba\w\w\w\w\w\w\b 匹配单词 “anterior” 的方式

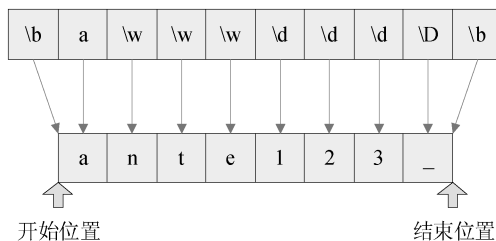


图 1.6 \b\w\w\w\w\d\d\d\D\b 匹配字符串 “ante123_” 的方式

1.3.3 元字符总结

正则表达式的常用元字符有`^`、`$`、`\b`、`.`、`\w`、`\W`、`\s`、`\S`、`\d` 和`\D`，它们的功能描述说明如表 1.1 所示。

表 1.1 常用元字符

字符	说明
<code>^</code>	匹配行的开始位置
<code>\$</code>	匹配行的结束位置
<code>\b</code>	匹配单词的开始或结束位置
<code>.</code>	匹配除换行符之外的任意字符
<code>\w</code>	匹配单词字符（包括字母、数字、下画线和汉字）
<code>\W</code>	匹配任意的非单词字符（包括字母、数字、下画线和汉字）
<code>\s</code>	匹配任意的空白字符，如空格、制表符、换行符、中文全角空格等
<code>\S</code>	匹配任意的非空白字符
<code>\d</code>	匹配任意的数字
<code>\D</code>	匹配任意的非数字字符

元字符`.`能够匹配除换行符之外的任意字符，如大写字母、小写字母、数字、`_`（下画线）等。以下正则表达式匹配除换行符之外的以任何字符分割字符串“2007”、“06”、“22”的字符串。

2007.06.22 (13)

元字符`\W`能够匹配除单词字符之外的任意字符。以下正则表达式匹配长度为 2 的字符串，且该字符串不包括单词字符。

\W\W (14)

使用工具 `Regex Tester` 测试正则表达式 2007.06.22，结果如图 1.7 所示。使用工具 `Regex Tester` 测试正则表达式`\W\W`，结果如图 1.8 所示。在图 1.8 所示结果中，匹配了 3 个结果：“?”、“*”和“**”。在第一个结果中，正则表达式`\W\W`中的第一个`\W`匹配字符“?”的上一行的换行符号，第二个`\W`才匹配字符“?”。在第三个结果中，正则表达式`\W\W`中的每一个`\W`都匹配字符“*”。

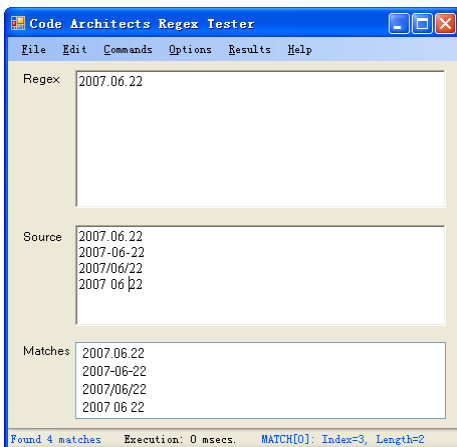


图 1.7 测试正则表达式 2007.06.22

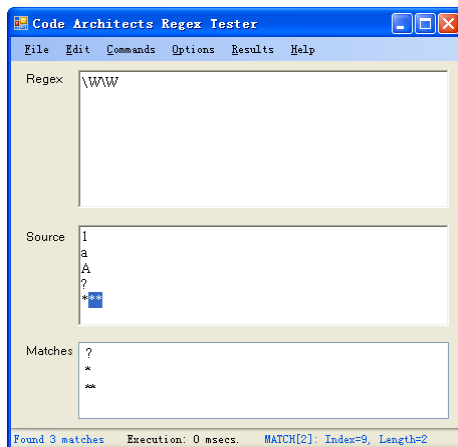


图 1.8 测试正则表达式 \W\W

元字符 `\s` 能够匹配空白字符，如空格、制表符、换行符、中文全角空格等。以下正则表达式首先匹配一个单词字符，然后匹配一个空白字符，最后匹配一个单词字符。

`\w\s\w` (15)

元字符 `\S` 能够匹配非空白字符，即除空格、制表符、换行符、中文全角空格等字符之外的字符。以下正则表达式首先匹配一个非空白字符，然后匹配一个空白字符，最后匹配一个非空白字符。

`\S\s\S` (16)

使用工具 `Regex Tester` 分别测试正则表达式 `\w\s\w` 和 `\S\s\S`，结果分别如图 1.9 和图 1.10 所示。`\S` 和 `\w` 都能够匹配单词字符，但是，`\S` 能够匹配除了单词字符之外的字符，如字符 `“/”` 和 `“*”` 等。

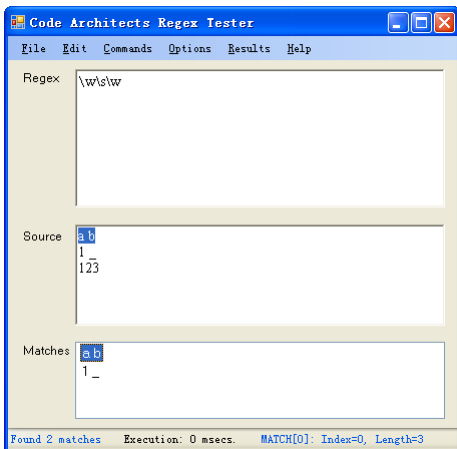


图 1.9 测试正则表达式 \w\s\w

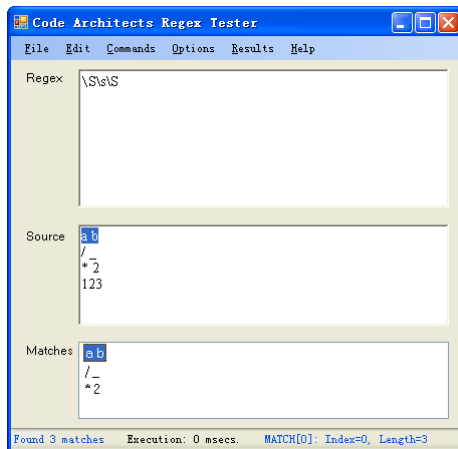


图 1.10 测试正则表达式 \S\s\S

元字符 `\d` 能够匹配 `0~9` 中的任意数字。以下正则表达式匹配 1 位的整数（即小于 10 的整数）。 (17)

以下正则表达式匹配 3 位的整数（即大于 99 小于 1000 的整数）。 (18)

`\d\d\d` (18)

元字符 `\D` 能够匹配除 `0~9` 之外的任何字符。以下正则表达式匹配以数字开头、非数字字符结尾的字符串，且数字字符是该字符串的第一个字符。

`\b\d\D` (19)

使用工具 Regex Tester 分别测试正则表达式 `\d\d\d` 和 `\b\dD`, 结果分别如图 1.11 和图 1.12 所示。

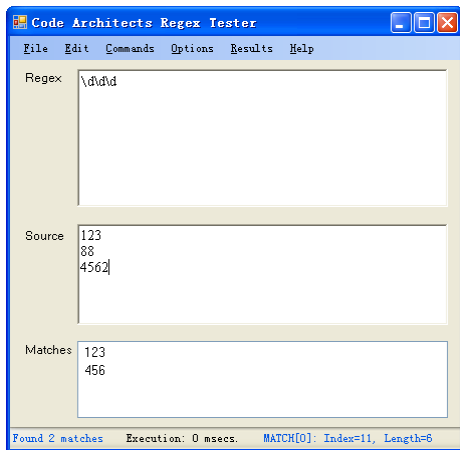


图 1.11 测试正则表达式 `\d\d\d`

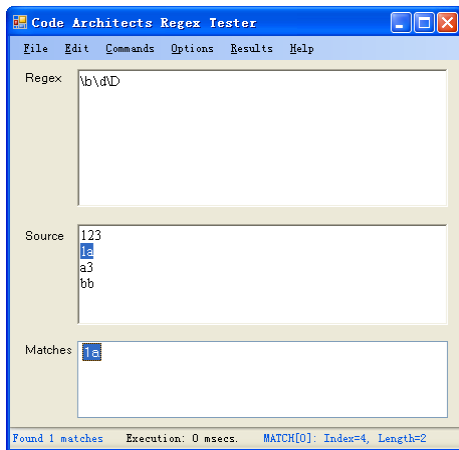


图 1.12 测试正则表达式 `\b\dD`

1.4 文字匹配

1.4.1 字符类

在正则表达式中, 元字符通常一次只能匹配一个位置或字符集合中的一个字符。通常情况下, 如果要匹配数字、字母、空白等字符时, 可以直接使用与这些集合相对应的元字符。然而, 如果要匹配的字符集合 (如集合 `[0,1,2,3,4,5]`) 没有与之相对应的元字符时, 则需要自定义匹配的字符集合。此时, 可以使用字符类解决这个问题。字符类是一个字符集合, 如果该字符集合中的任何一个字符被匹配, 则它就会找到该匹配项。

字符类是正则表达式中的“迷你”语言, 可以在方括号“`[]`”中定义。最简单的字符类由方括号“`[]`”和一个字母表构成, 如元音字符类 `[aeiou]`。以下正则表达式匹配数字 0、1、2、3、4、5、6 中的任何一个。

```
[0123456] (20)
```

以下正则表达式可以匹配任何数字 (即 0、1、2、3、4、5、6、7、8、9)。

```
[0123456789] (21)
```

以下正则表达式匹配 HTML 标记中的“`<H1>`”、“`<H2>`”、“`<H3>`”、“`<H4>`”、“`<H5>`”或“`<H6>`”。

```
<H[123456]> (22)
```

以下正则表达式匹配字符串“Jack”或者“jack”。

```
[Jj]ack (23)
```

然而, 正则表达式 `[0123456789]` 的书写非常不方便。因此, 正则表达式引入了连接符“-”来定义字符的范围。以下正则表达式等价于正则表达式 `[0123456789]`。

```
[0-9] (24)
```

以下正则表达式可以匹配任意小写字母。

```
[a-z] (25)
```

以下正则表达式可以匹配任意大写字母。

```
[A-Z] (26)
```

注意：当且仅当在字符类中的连接符“-”不是第一个字符时，它才具有特殊的含义：它可以指定字符类的最大边界和最小边界之间的任何字符。它的具体含义由具体的字符类决定。因此，字符类的最大边界和最小边界，以及字符在 ASCII 或 Unicode 表中出现的顺序共同确定了连接符“-”指定的字符的范围。

如果要在字符类中包括连接符“-”，则必须将它作为第一个字符。如正则表达式`[-a]`匹配字符“-”或者“a”。

在字符类中，若字符“^”是字符类的第一个字符，则表示否定该字符类，即匹配除了该字符类之外的任意字符。以下正则表达式可以匹配任何非元音字符。

```
[^aAeEiIoOuU] (27)
```

以下正则表达式可以匹配除连接符“-”之外的任何字符。

```
[^-] (28)
```

以下正则表达式匹配字符 a 之后不是字符 b 的字符串。

```
a[^b] (29)
```

使用工具 Regex Tester 测试正则表达式 `a[^b]` 的结果如图 1.13 所示。

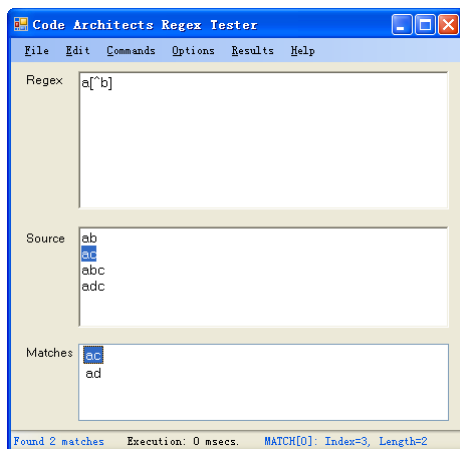


图 1.13 测试正则表达式 `a[^b]`

注意：正则表达式中的元字符在字符类中不做任何特殊处理，它仅仅代表一个自身的字符。如正则表达式`[-.]`只能匹配字符“-”和“.”，它不能匹配除字符“-”和“.”之外的任何字符。因此，在字符类中使用元字符时，不需要使用转义运算。

在正则表达式中，常用的字符类如表 1.2 所示。

表 1.2 常用的字符类

字符或表达式	说明
.	匹配除换行符之外的任意字符
\w	匹配单词字符（包括字母、数字、下画线和汉字）
\W	匹配任意的非单词字符（包括字母、数字、下画线和汉字）
\s	匹配任意的空白字符，如空格、制表符、换行符、中文全角空格等
\S	匹配任意的非空白字符
\d	匹配任意的数字
\D	匹配任意的非数字字符
[aeiou]	匹配字符集合中的任何字符

续表

字符或表达式	说明
[^aeiou]	匹配除字符集合中的以外的字符
[0-9a-zA-Z_]	匹配任何数字、字母（大写字母和小写字母）和下画线，等同于\w
[^0-9a-zA-Z_]	匹配除任何数字、字母、下画线之外的任何字符，等同于\W
\p{name}	匹配{name}指定的命名字符类中的任何字符
\P{name}	匹配除{name}指定的命名字符类中之外的任何字符

注意：在表 1.2 中，表达式\p{name}和\P{name}为 .NET Framework 所支持。

1.4.2 字符转义

正则表达式定义了一些特殊的元字符，如^、\$、.等。由于这些字符在正则表达式中被解释成其他的指定的意义，如果需要匹配这些字符，则需要使用字符转义来解决这一问题。转义字符为“\”（反斜杠），它可以取消这些字符（如^、\$、.等）在表达式中具有的特殊意义。

以下正则表达式匹配字符“.”。

```
\. (30)
```

以下正则表达式匹配字符“*”。

```
\* (31)
```

以下正则表达式匹配字符“\”。

```
\\ (32)
```

以下正则表达式匹配字符串“www.myweburl.com”。

```
www\\.myweburl\\.com (33)
```

正则表达式的常用转义字符的说明如表 1.3 所示。其中，除.、\$、^、{、[、(、|、)、*、+、?、\之外的字符不需要进行转义，它们都表示字符本身。

表 1.3 常用字符转义

字符或表达式	说明
\a	响铃（警报）\u0007
\b	在正则表达式中，表示单词的边界；如果在字符类中，则表示退格符\u0008
\t	制表符\u0009
\r	回车符\u000D
\v	垂直制表符\u000B
\f	换页符\u000C
\n	换行符\u000A
\e	回退（Esc）符\u001B
\040	将ASCII字符匹配为八进制数（最多三位）
\x20	使用十六进制表示，形式与ASCII字符匹配
\cC	ASCII控制字符，如Ctrl-C
\u0020	使用十六进制表示形式（恰好四位）与Unicode字符匹配

1.4.3 反义

在使用正则表达式时，如果需要匹配不在字符类指定范围内的字符时，可以使用反义规则。

以下正则表达式匹配字符 a 之后不是字符 b 的字符串。

```
a[^b] (34)
```

以下正则表达式匹配被尖括号括起来的、以字符串“asp”开头的、倒数第二个字符不能为字符“>”的、长度为6的任意字符串。

<asp[^>]>

(35)

其实，在前面的小节中已经使用了反义的表达式，如\W、\S、\D、[^aeiou]等。常用的反义表达式如表1.4所示。

表 1.4 常用的反义表达式

字符或表达式	说明
\W	匹配任意的非单词字符（包括字母、数字、下画线和汉字）
\S	匹配任意的非空白字符
\D	匹配任意的非数字字符
\B	匹配不是单词开头和结束的任何位置
[^a]	匹配除字符a之外的任意字符
[^aeiou]	匹配除字符集合（aeiou）中的字符之外的任意字符

1.4.4 限定符

正则表达式的元字符一次一般只能匹配一个位置或一个字符，如果想要匹配零个、一个或多个字符时，则需要使用限定符。限定符用于指定允许特定字符或字符集自身重复出现的次数。如{n}表示重复n次、{n,}表示重复至少n次、{n,m}表示重复至少n次，最多m次。常用限定符的说明如表1.5所示。

表 1.5 常用限定符

字符或表达式	说明
{n}	重复n次
{n,}	重复至少n次
{n,m}	重复至少n次，最多m次
*	重复至少0次，等同于{0,}
+	重复至少1次，等同于{1,}
?	重复0次或1次，等同于{0,1}
*?	尽可能少地使用重复的第一个匹配
+?	尽可能少地使用重复但至少使用一次
??	使用零次重复（如有可能）或一次重复
{n}?	等同于{n}
{n,}?	尽可能少地使用重复，但至少使用n次
{n,m}?	介于n次和m次之间、尽可能少地使用重复

以下正则表达式可以匹配字符串“color”或者“colour”。表达式u?表示字母“u”可以出现1次或者不出现。

colou?r

(36)

以下正则表达式可以匹配字符串“four”或者“for”。表达式u?表示字母“u”可以出现1次或者不出现。

fou?r

(37)

以下正则表达式匹配以字符串“name”开头的、以数字字符串结尾的字符串。其中，表达式\d+可以匹配长度至少为1的数字字符串。

\bname\d+\b

(38)

以下正则表达式匹配被尖括号括起来的、以字符串“asp:TextBox ”（最后一个字符是空格）

开头的字符串。正则表达式`<asp:TextBox [^>]+>`中的字符类`[^>]`匹配除了尖括号“>”之外的任何字符。

`<asp:TextBox [^>]+>` (39)

以下正则表达式匹配以字母 a 开头的单词。`\w` 匹配一个单词字符，`\w*` 表示该单词字符可以重复零次或多次。

`\ba\w*\b` (40)

正则表达式`\ba\w*\b`匹配单词“anterior”的过程如图 1.14 所示，匹配的具体步骤如下。

- ① 匹配单词的开始位置；
- ② 在步骤①时匹配字符 a；
- ③ 匹配字符 a 之后，该表达式可以直接通过步骤②匹配到单词的结束位置，并完成整个匹配过程，从而匹配单词“a”；
- ④ 该表达式在匹配字符 a 之后，可以通过步骤③来匹配任意单词字符；
- ⑤ 重复步骤④一次或多次，即匹配一个或多个任意单词字符；
- ⑥ 匹配到单词的结束位置，并完成整个匹配过程，从而匹配到以字符 a 开头的长度大于 1 的单词。

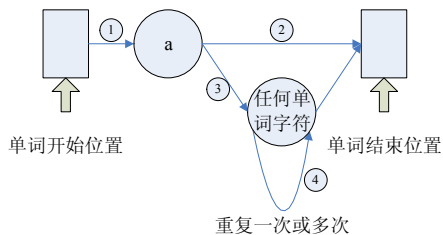


图 1.14 `\ba\w*\b` 匹配过程

限定符除了使用*、+和?之外，还可以使用数字直接指定重复的次数。`{n}`表示重复 n 次；`{n,}`表示至少重复 n 次，而最多重复的次数不限制；`{n,m}`（其中， $n \leq m$ ）表示至少重复 n 次、最多重复 m 次。以下正则表达式匹配 3 位的整数，该正则表达式和正则表达式`\d\d\d`等价。

`\d{3}` (41)

以下正则表达式匹配 3 位以上（包括 3 位）的整数，该正则表达式和正则表达式`\d\d\d\d*`或`\d\d\d+`等价。

`\d{3,}` (42)

以下正则表达式匹配 3 位或 4 位的整数，该正则表达式和正则表达式`\d\d\d\d?`等价。

`\d{3,4}` (43)

以下正则表达式匹配当前国内以“13”开头的手机号码：即以字符串“13”开头的、后接连续 9 个数字的字符串。

`13\d{9}` (44)

以下正则表达式匹配当前国内部分地区的固定电话号码。其中，号码的前 3 位为区号，后 8 位为本地号码，区号和本地号码使用连接符号“-”进行连接。

`0\d{2}-\d{8}` (45)

以下正则表达式匹配当前国内部分地区的固定电话号码。其中，号码的前 4 位为区号，后 7 位为本地号码，区号和本地号码使用连接符号“-”进行连接。

`0\d{3}-\d{7}` (46)

以下正则表达式能够匹配满足以下特征的单词。

- ❑ 字符串只包含字符 a 和 b，且字符 b 必须在字符 a 之后；
- ❑ 字符 a 连续的个数最多为 4，最小为 1；
- ❑ 字符 b 连续的个数最多为 3，最小为 1。

`\ba{1,4}b{1,3}\b` (47)

正则表达式 `\ba{1,4}b{1,3}\b` 能够匹配的单词如下：

ab、aab、aaab、aaaab、abb、abbb、aabb、aabbb、aaabb、aaabbb、aaaabb、aaaabbb。

使用工具 Regex Tester 测试正则表达式 `\ba{1,4}b{1,3}\b` 的结果如图 1.15 所示。

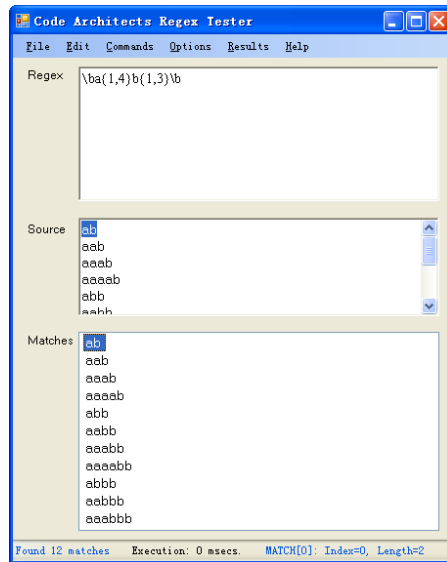


图 1.15 测试正则表达式 `\ba{1,4}b{1,3}\b`

如果在限定符 `*`、`+`、`?`、`{n}`、`{n,}` 和 `{n,m}` 之后再添加一个字符 “`?`”，则表示尽可能少地重复字符 “`?`” 之前的限定符号的重复次数，这种方式匹配被称为懒惰匹配。与之相对应的是贪婪匹配，即仅仅使用单个限定符 `*`、`+`、`?`、`{n}`、`{n,}` 和 `{n,m}` 的匹配。常用的懒惰限定符如表 1.6 所示。

表 1.6 常用的懒惰限定符

字符或表达式	说明
<code>*?</code>	尽可能少地使用重复的第一个匹配
<code>+?</code>	尽可能少地使用重复，但至少使用一次
<code>??</code>	使用零次重复（如有可能）或一次重复
<code>{n}?</code>	等同于 <code>{n}</code>
<code>{n,}?</code>	尽可能少地使用重复但至少使用 n 次
<code>{n,m}?</code>	介于 n 次和 m 次之间、尽可能少地使用重复

以下正则表达式匹配以字母 a 开头、以字母 b 结束的最长字符串。此时，这是一种贪婪匹配。

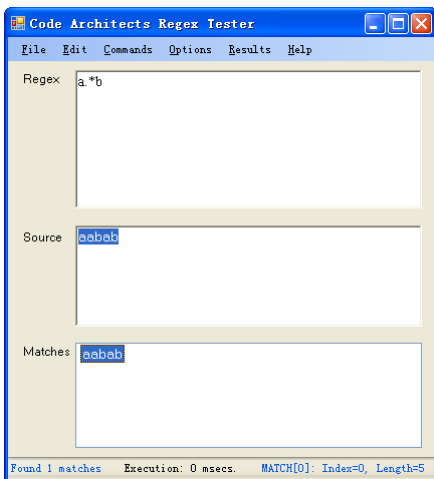
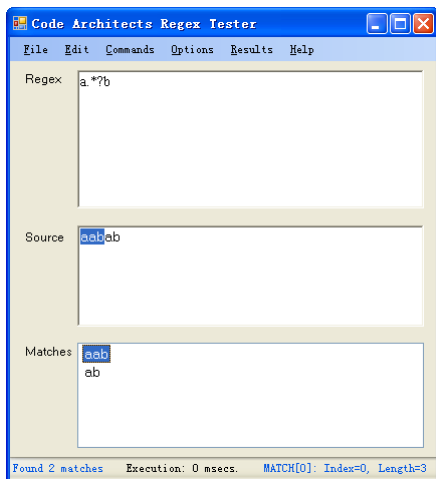
`a.*b` (48)

以下正则表达式匹配以字母 a 开头、以字母 b 结束的最短字符串。此时，这是一种懒惰匹配。

`a.*?b` (49)

如果将正则表达式 `a.*b` 应用于字符串 “aabab”，则匹配字符串 “aabab”。如果将正则表达式 `a.*?b` 应用于字符串 “aabab”，则匹配字符串 “aab” 和字符串 “ab”，而不会匹配字符串 “aabab”。

使用工具 Regex Tester 分别测试正则表达式 `a.*b` 和 `a.*?b`，结果分别如图 1.16 和图 1.17 所示。

图 1.16 测试正则表达式 `a.*b`图 1.17 测试正则表达式 `a.*?b`

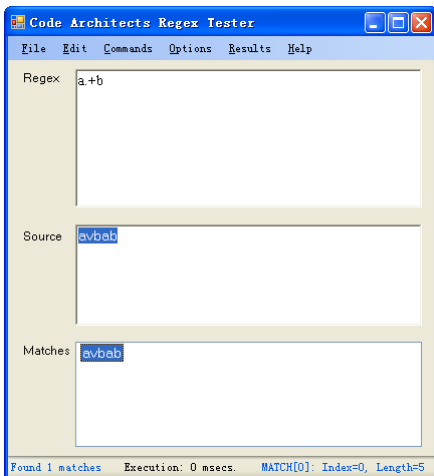
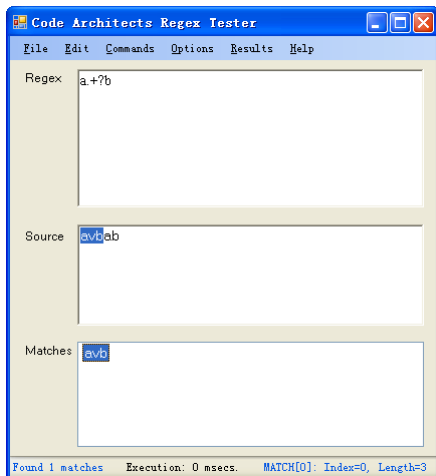
以下正则表达式匹配以字母 `a` 开头、以字母 `b` 结束、长度至少为 3 的字符串。此时，这是一种贪婪匹配。

`a.+b` (50)

以下正则表达式匹配以字母 `a` 开头、以字母 `b` 结束、长度至少为 3 的字符串。此时，这是一种懒惰匹配。

`a.+?b` (51)

正则表达式 `a.+?b` 在匹配过程中，字母 `a` 和字母 `b` 之间的字符串实际上只重复了 1 次。如果将正则表达式 `a.+b` 应用于字符串“avbab”，则匹配字符串“avbab”。如果将正则表达式 `a.+?b` 应用于字符串“avbab”，则匹配字符串“avb”，而不会匹配字符串“avbab”。使用工具 Regex Tester 分别测试正则表达式 `a.+b` 和 `a.+?b`，结果分别如图 1.18 和图 1.19 所示。

图 1.18 测试正则表达式 `a.+b`图 1.19 测试正则表达式 `a.+?b`

以下正则表达式匹配以字母 `a` 开头、以字母 `b` 结束、长度为 2 或者 3 的字符串。此时，这是一种贪婪匹配。

`a.?b` (52)

以下正则表达式匹配以字母 `a` 开头、以字母 `b` 结束、长度为 3 的字符串。此时，这是一种懒

惰匹配。

`a.??b` (53)

正则表达式 `a.??b` 在匹配过程中，字母 `a` 和字母 `b` 之间的字符要么出现，要么最多出现 1 次。使用工具 `Regex Tester` 分别测试正则表达式 `a.?b` 和 `a.??b`，结果分别如图 1.20 和图 1.21 所示。

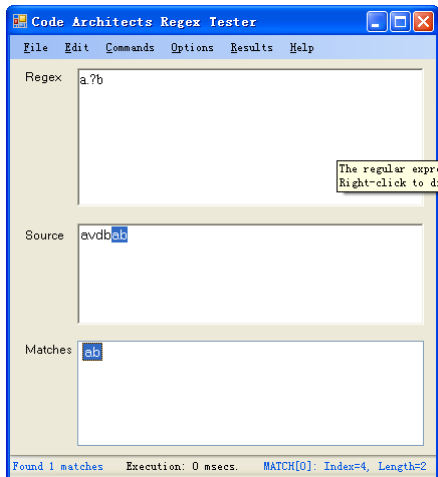


图 1.20 测试正则表达式 `a.?b`

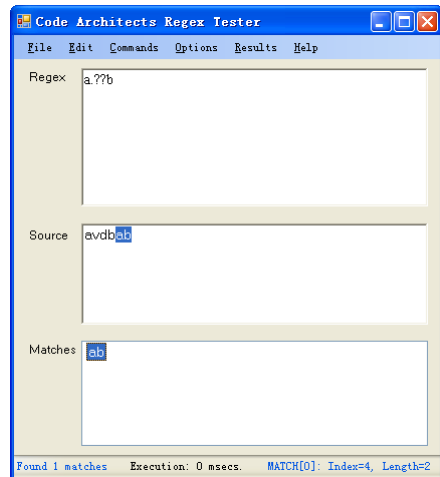


图 1.21 测试正则表达式 `a.??b`

以下正则表达式匹配以字母 `a` 开头、以字母 `b` 结束、长度至少为 3 的字符串。此时，这是一种贪婪匹配。

`a.{1,}b` (54)

以下正则表达式匹配以字母 `a` 开头、以字母 `b` 结束、长度至少为 3 的字符串。此时，这是一种懒惰匹配。

`a.{1,}?b` (55)

正则表达式 `a.{1,}?b` 在匹配过程中，字母 `a` 和字母 `b` 之间的字符串实际上只重复了 1 次。如果将正则表达式 `a.{1,}b` 应用于字符串“avbab”，则匹配字符串“avbab”。如果将正则表达式 `a.{1,}?b` 应用于字符串“avbab”，则匹配字符串“avb”，而不会匹配字符串“avbab”。使用工具 `Regex Tester` 分别测试正则表达式 `a.{1,}b` 和 `a.{1,}?b`，结果分别如图 1.22 和图 1.23 所示。

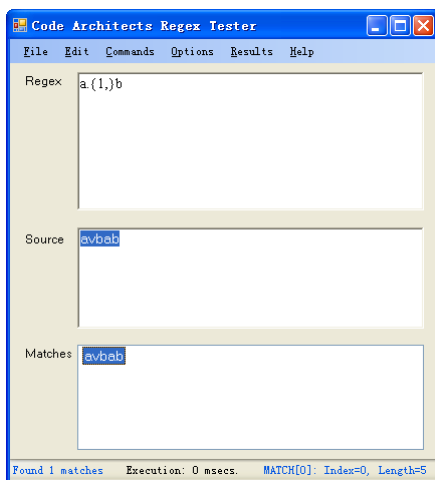


图 1.22 测试正则表达式 `a.{1,}b`

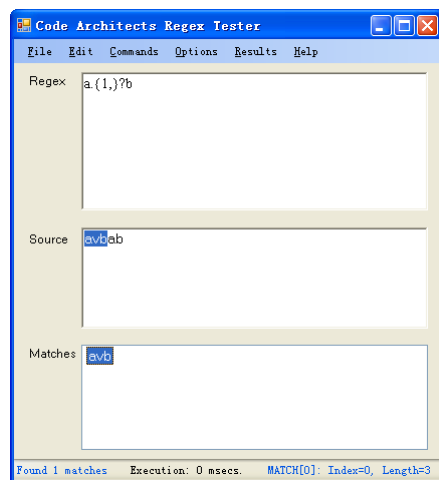


图 1.23 测试正则表达式 `a.{1,}?b`

1.5 字符的运算

1.5.1 替换

正则表达式 `0\d{2}-\d{8}` 和 `0\d{3}-\d{7}` 分别匹配区号为 3 位和 4 位的固定电话号码, 如果需要同时匹配区号为 3 位和 4 位的固定电话号码时, 使用替换可以满足这一需求。最简单的替换是使用字符 “|” 表示, 它表示如果某一个字符串匹配了正则表达式中的字符 “|” 的左边或者右边的规则, 那么该字符串也匹配了该正则表达式。

以下正则表达式匹配了当前国内部分地区的两种固定电话号码: 一种是号码的前 4 位为区号, 后 7 位为本地号码; 另一种是号码的前 3 位为区号, 后 8 位为本地号码。其中, 区号和本地号码都使用连字符 “-” 进行连接。

```
0\d{2}-\d{8}|0\d{3}-\d{7} (56)
```

以下正则表达式匹配了当前国内部分地区的三种固定电话号码: 一种是号码的前 3 位为区号, 后 8 位为本地号码; 另一种是号码的前 4 位为区号, 后 7 位为本地号码; 最后一种是号码的前 4 位为区号, 后 8 位为本地号码。其中, 区号和本地号码都使用连字符 “-” 进行连接。

```
0\d{2}-\d{8}|0\d{3}-\d{7}|0\d{3}-\d{8} (57)
```

以下正则表达式匹配当前国内部分地区的区号为 4 位的固定电话号码。其中, 区号和本地号码可以使用连字符 “-” 进行连接, 也可以不使用连接符号 “-”。

```
0\d{3}-\d{7}|0\d{3}[-]? \d{7} (58)
```

使用工具 **Regex Tester** 测试正则表达式 `0\d{3}-\d{7}|0\d{3}[-]? \d{7}`, 测试结果如图 1.24 所示。其中, 电话号码 0731-1234567 和 07311234567 均被匹配。

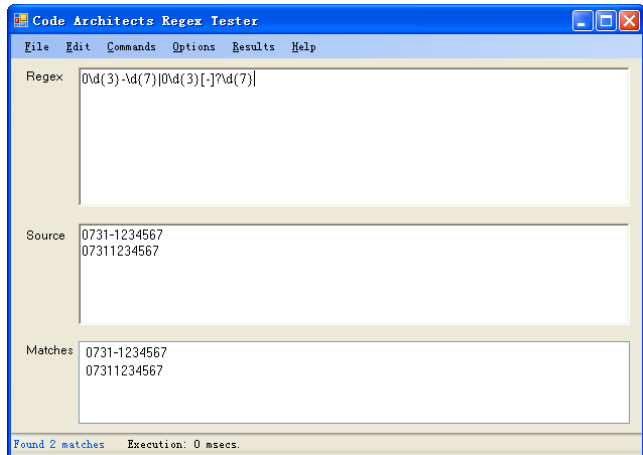


图 1.24 测试 `0\d{3}-\d{7}|0\d{3}[-]? \d{7}` 匹配电话号码

正则表达式 `[Jj]ack` 可以匹配字符串 “Jack” 或者 “jack”。该正则表达式还可以使用替换来实现同样的匹配效果。以下正则表达式等效于正则表达式 `[Jj]ack`。

```
Jack|jack (59)
```

以下正则表达式等效于正则表达式 `Jack|jack`。因此, 正则表达式 `[Jj]ack`、`Jack|jack` 和 `Jack|jack` 能够匹配的所有字符串都是相同的。

```
(J|j)ack (60)
```

使用工具 **Regex Tester** 分别测试正则表达式 `[Jj]ack` 和 `(J|j)ack`, 结果分别如图 1.25 和图 1.26 所示。从图中可以看到, 两个正则表达式匹配的结果是相同的。

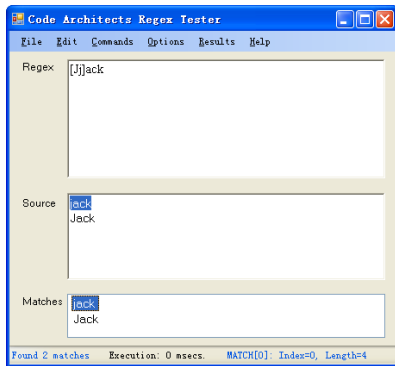


图 1.25 测试正则表达式[J]jack

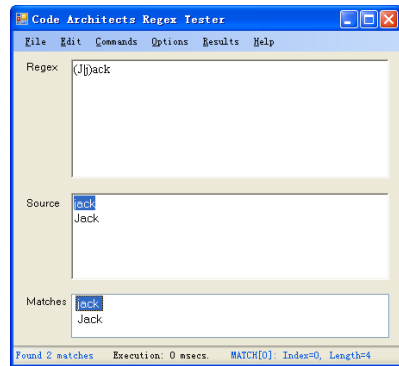


图 1.26 测试正则表达式(J|j)ack

正则表达式的常用替换说明如表 1.7 所示。

表 1.7 正则表达式的常用替换

字符或表达式	说明
	匹配 （竖线）字符的左侧或右侧
(?(表达式)yes no)	表达式要么与“yes”部分匹配；要么与“no”部分匹配。其中，“no”部分可省略
(?(name)yes no)	以name命名的字符串要么与“yes”部分匹配，要么与“no”部分匹配。其中，“no”部分可省略

注意：字符|在匹配表达式时，首先匹配|字符的左侧部分，当左侧部分不匹配时，它才尝试匹配|字符的右侧部分。

有以下两个正则表达式：

`\d{5}-\d{3}|\d{5}` (61)

`\d{5}|\d{5}-\d{3}` (62)

根据字符|的匹配原则（优先匹配左侧表达式），正则表达式`\d{5}|\d{5}-\d{3}`只能匹配 5 位的数字字符串，而不会匹配用连接符号连接的 8 位数字字符串。然而，正则表达式`\d{5}-\d{3}|\d{5}`能够匹配用连接符号连接的 8 位数字字符串或者 5 位的数字字符串。因为，该表达式首先尝试匹配用连接符号连接的 8 位数字字符串，只有当未匹配时，才匹配 5 位的数字字符串。

使用工具 Regex Tester 分别测试了正则表达式`\d{5}|\d{5}-\d{3}`和`\d{5}-\d{3}|\d{5}`，测试结果分别如图 1.27 和图 1.28 所示。正则表达式`\d{5}|\d{5}-\d{3}`只匹配了字符串“12345”。而正则表达式`\d{5}-\d{3}|\d{5}`可以匹配字符串“12345-678”和字符串“12345”。

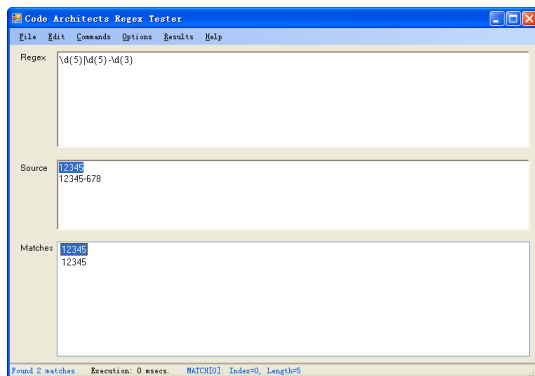


图 1.27 测试正则表达式\d{5}|\d{5}-\d{3}

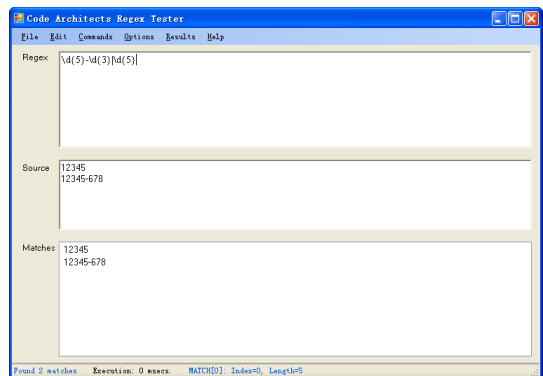


图 1.28 测试正则表达式\d{5}-\d{3}|\d{5}

1.5.2 分组

分组又称为子表达式，即把一个正则表达式的全部或部分分成一个或多个组。其中，分组使用的字符为“(”和“)”，即左圆括号和右圆括号。分组之后，可以将“(”和“)”之间的表达式看成一个整体来处理。以下正则表达式可以匹配重复出现字符串“abc”一次或两次的字符串。此时，表达式将“abc”看成一个整体来进行重复匹配。

```
(abc){1,2} (63)
```

以下正则表达式可以匹配简单的 IP 地址。

```
(\d{1,3}\.){3}\d{1,3} (64)
```

正则表达式解释如下。

- 表达式 `\d{1,3}` 先匹配 1~3 位的整数，然后匹配一个字符“.”（点号），如“1.”、“12.”、“123.”、“888.”等。
- 表达式 `(\d{1,3}\.){3}` 将子表达式 `\d{1,3}` 匹配的字符串重复 3 次，如“1.2.3.”、“12.34.56.”、“123.456.789.”、“888.899.569.”等。
- 表达式 `\d{1,3}` 将匹配 1~3 位的整数。

综合以上分析，正则表达式 `(\d{1,3}\.){3}\d{1,3}` 能够匹配简单的 IP 地址，如“10.0.0.1”、“123.123.235.235”等。使用工具 Regex Tester 测试正则表达式 `(\d{1,3}\.){3}\d{1,3}`，结果如图 1.29 所示。

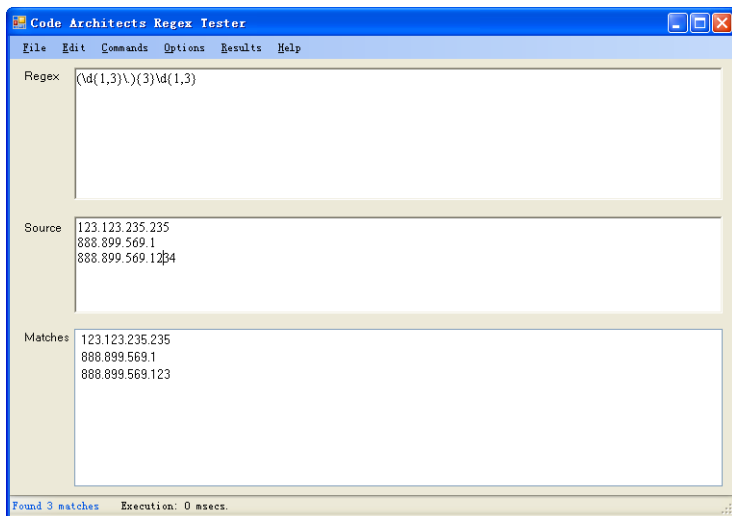


图 1.29 测试正则表达式 `(\d{1,3}\.){3}\d{1,3}`

1.5.3 反向引用

当一个正则表达式被分组之后，每一个组将自动被赋予一个组号，该组号可以代表该组的表达式。其中，组号的编制规则为：从左到右、以分组的左括号“(”为标志，第一个分组的组号为 1，第二个分组的组号为 2，依此类推。

反向引用提供了查找重复字符组的简便方法。它们可以被认为是再次匹配同一个字符串的快捷指令。反向引用可以使用数字命名（即默认名称）的组号，也可以使用指定命名的组号。具体说明如表 1.8 所示。

表 1.8 反向引用

表达式	说明
\数字	使用数字命名的反向引用
\k<name>	使用指定命名的反向引用

注意：在表 1.8 中，表达式 \k<name> 为 .NET Framework 所支持。

以下正则表达式匹配具有两个重复字符的单词。

`\b(\w)\1\b` (65)

以下正则表达式首先匹配单词的开头处，然后匹配一个字符和数字，再重复该字符和数字，最后是单词的结尾处。

`\b(\w)(\d)\1\2\b` (66)

注意：正则表达式 `\b(\w)\1\b` 和 `\b(\w)\w\b` 并不等效，第一个表达式只匹配含有两个相同字符的单词，而第二个表达式则匹配具有两个字符（可以相同，也可以不相同）的单词。

使用工具 **Regex Tester** 分别测试正则表达式 `\b(\w)\w\b` 和 `\b(\w)\1\b`，结果分别如图 1.30 和图 1.31 所示。在测试结果中，正则表达式 `\b(\w)\1\b` 只匹配单词“aa”，而正则表达式 `\b(\w)\w\b` 可以匹配单词“aa”和“ab”。

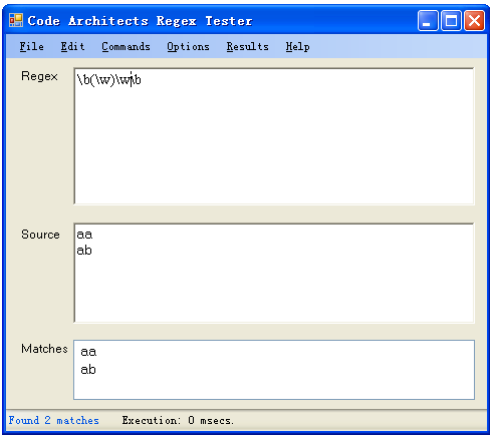


图 1.30 测试正则表达式 `\b(\w)\1\b`

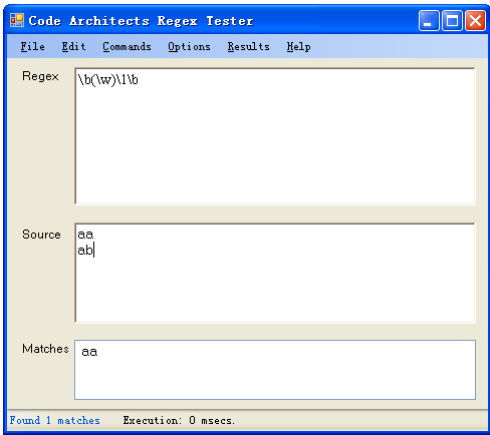


图 1.31 测试正则表达式 `\b(\w)\w\b`

以下正则表达式匹配以两个重复字符结尾的单词。

`\b\w*(\w+)\1\b` (67)

以下正则表达式匹配重复出现的单词。

`\b(\w+)\b\s+\1\b` (68)

上述正则表达式 `\b(\w+)\b\s+\1\b` 匹配的具体过程如下。

- ① 表达式 `\b(\w+)\b` 匹配一个单词，且单词的长度至少为 1。
- ② 表达式 `\s+` 匹配一个或多个空白字符。
- ③ 表达式 `\1` 将重复子表达 `(\w+)` 匹配的内容，即重复匹配的单词。
- ④ 匹配单词的结束位置。

分组不但可以使用数字作为组号，而且还可以使用自定义名称作为组号。以下两个正则表达式都是将分组后的子表达式 `\w+` 命名为“word”。

```
(?<word>\w+) (69)
(? 'word' \w+) (70)
```

因此，正则表达式**`\b(\w+)\b\s+\l\b`**和以下正则表达式等价，它们都匹配重复出现的单词。

```
\b(?<word>\w+)\b\s+\k<word>\b (71)
```

以下正则表达式和正则表达式**`\b\w*(\w+)\l\b`**等价，也是匹配以两个重复字符结尾的单词。

```
\b\w*(?<char>\w+)\k<char>\b (72)
```

分组子表达式**`(?<name>)`**将元字符括在其中，并强制正则表达式引擎记住该子表达式匹配，同时使用“**name**”将该匹配进行命名。反向引用**`\k<name>`**使引擎对当前字符和以名称“**name**”存储的先前匹配字符进行比较，从而匹配具有重复字符的字符串。正则表达式中的常用分组说明如表 1.9 所示。

表 1.9 常用分组说明

字符	说明
(experssion)	匹配字符串 experssion ，并将匹配的文本保存到自动命名的组里
(? <name> experssion)	匹配字符串 experssion ，并将匹配的文本以 name 进行命名。该名称不能包含标点符号，不能以数字开头
(?: experssion)	匹配字符串 experssion ，不保存匹配的文本，也不给此组分配组号
(? = experssion)	匹配字符串 experssion 前面的位置
(? ! experssion)	匹配后面不是字符串 experssion 的位置
(? <= experssion)	匹配字符串 experssion 后面的位置
(? <! experssion)	匹配前面不是字符串 experssion 的位置
(? > experssion)	只匹配字符串 experssion 一次

1.6 正则的其他运算

1.6.1 零宽度断言

在前面的小节中，元字符**`\b`**、**`^`**和**`$`**都匹配一个位置，且这个位置满足一定的条件。这里，把满足的这一个条件称为断言或零宽度断言。正则表达式中的常用零宽度断言如表 1.10 所示。

表 1.10 零宽度断言

字符（断言）	说明
<code>^</code>	匹配行的开始位置
<code>\$</code>	匹配行的结束位置
<code>\A</code>	匹配必须出现在字符串的开头
<code>\Z</code>	匹配必须出现在字符串的结尾或字符串结尾处的换行符号 <code>n</code> 之前
<code>\z</code>	匹配必须出现在字符串的结尾
<code>\G</code>	匹配必须出现在上一个匹配结束的地方
<code>\b</code>	匹配字符的开始或结束位置
<code>\B</code>	匹配不是在字符的开始或结束位置

在表 1.9 中，表达式**`(?=experssion)`**、**`(?!experssion)`**、**`(?<=experssion)`**和**`(?<!experssion)`**都是匹配一个位置。下面将详细介绍表达式**`(?=experssion)`**和**`(?<=experssion)`**。

`(?=experssion)`又称为零宽度正预测先行断言，它断言自身位置的后面能够匹配表达式**experssion**。以下正则表达式匹配以字符串“**ed**”结尾的单词的前面部分，即匹配单词的除字符串“**ed**”之外的部分。

```
\b\w+(?=ed\b)
```

(73)

(?<=experssion)又称为零宽度正回顾后发断言，它断言自身位置的前面能够匹配表达式 experssion。以下正则表达式匹配以字符串“an”开头的单词的后面部分，即匹配单词的除字符串“an”之外的部分。

```
(?<=\ban)\w+\b
```

(74)

使用工具 **Regex Tester** 分别测试正则表达式 `\b\w+(?=ed\b)` 和 `(?<=\ban)\w+\b`，结果分别如图 1.32 和图 1.33 所示。

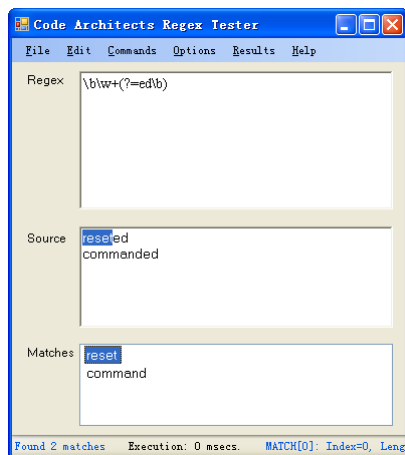


图 1.32 测试正则表达式 `\b\w+(?=ed\b)`

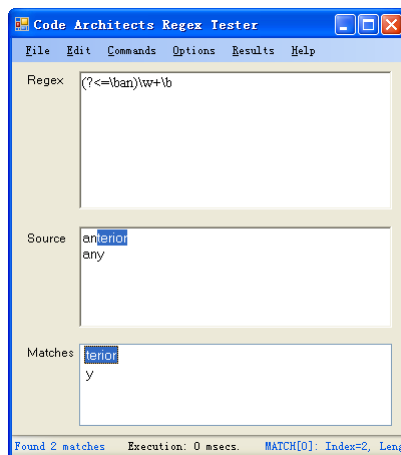


图 1.33 测试正则表达式 `(?<=\ban)\w+\b`

1.6.2 负向零宽度断言

零宽度断言只能指定或匹配一个位置。而负向零宽度断言与零宽度断言恰恰相反，它能够指定或匹配不止一个位置，即所说的“反义”。特别是在匹配字符串中不包含指定的字符时，负向零宽度断言特别有用。以下正则表达式中的表达式 `a(?!b)` 将断言字符“a”之后不能为字符“b”。

```
\b\w*a(?!b)\w*\b
```

(75)

因此，正则表达式 `\b\w*a(?!b)\w*\b` 匹配单词字符串，且该字符串中的字符“a”之后不能为字符“b”。

表达式 `(?!experssion)` 又称为负向零宽度断言或者零宽度负预测先行断言，它断言自身位置的后面不能匹配字符串 experssion。以下正则表达式首先匹配长度为 3 的单词字符串，该字符串之后不能是数字字符串。

```
\b\w{3}(?!d+)
```

(76)

使用工具 **Regex Tester** 测试正则表达式 `\b\w{3}(?!d+)`，结果如图 1.34 所示。

表达式 `(?<!\d+)` 又称为零宽度负回顾后发断言，它断言自身位置的前面不能匹配字符串 experssion。

以下正则表达式匹配不以数字开头的字符串，该字符串的开头只能包括大写字母、小写字母或下划线。

```
(?<!\d+)[a-z_A-Z] +
```

(77)

使用工具 **Regex Tester** 测试正则表达式 `(?<!\d+)[a-z_A-Z] +`，结果如图 1.35 所示。

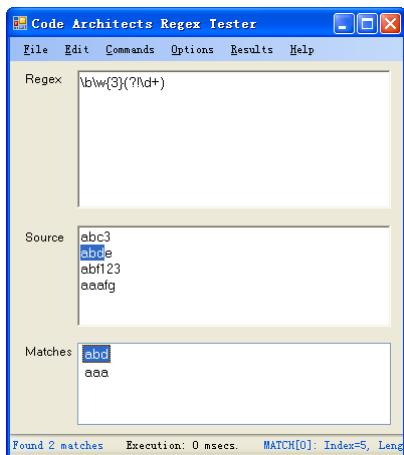


图 1.34 测试正则表达式\b\w{3}(?!d+)



图 1.35 测试正则表达式(?<!\d+)[a-z_A-Z]+

1.6.3 匹配选项

匹配选项可以指定正则表达式匹配中的行为，如忽略大小写、处理多行、处理单行、从右到左开始执行匹配等。

注意：本节中下面介绍的匹配选项为 .NET Framework 所支持。

.NET Framework 正则表达式中的常用匹配选项说明如表 1.11 所示。

表 1.11 常用匹配选项

选项	内联字符	说明
None	N/A	指定不设置任何选项
IgnoreCase	i	指定不区分大小写
Multiline	m	指定多行模式，即修改^和\$的含义，以使它们分别与任何行的开头和结尾匹配
ExplicitCapture	n	指定必须指定分组的名称或组号
Compiled	N/A	指定正则表达式将被编译为程序集
Singleline	s	指定单行模式
IgnorePatternWhitespace	x	指定消除表达式中空白字符，并启用字符(#)后面的注释
RightToLeft	N/A	指定匹配是从右向左而不是从左向右进行的
ECMAScript	N/A	指定已为表达式启用了符合ECMAScript的行为
CultureInvariant	N/A	指定忽略语言中的区域性差异

1.6.4 注释

正则表达式中除了表达式的基本内容外，还可以包括注释。其中，注释一般通过表达式 (?# 注释) 实现。以下正则表达式在第一个分组中添加了注释“不能以数字开头”。

```
(?<!\d+(?#不能以数字开头))[a-z_A-Z]+ (78)
```

注意：如果要在正则表达式中包含注释，则最好打开 IgnorePatternWhitespace 选项，即忽略模式里的空白字符。因此，此时可以在注释中添加空格、换行符号、制表符号等。一旦启用了该选项，则符号#之后的内容全部被忽略。

正则表达式(?<!\d+)[a-z_A-Z]+可以写成以下形式。

```
(?<!          # 断言是否能匹配该组中的表达式 (79)
\d+          # 长度至少为 1 的数字字符串
)           # 表达式结束
[a-z_ A-Z]+  # 只能包括大写字母、小写字母或下画线的长度至少为 1 的字符串
```

1.6.5 优先级顺序

正则表达式存在元字符、转义符、限定符、|等运算或表达式。在匹配过程中，正则表达式都事先规定了这些运算或表达式的优先级。正则表达式也可以像数学表达式一样来求值。也就是说，正则表达式可以从左至右、并按照一个给定的优先级来求值。表 1.12 是按照优先级从高到低的顺序列出的正则表达式运算符的优先级顺序表。

表 1.12 优先级顺序表

运算符或表达式	说明
\	转义符
()、(?:)、(?:=)、[]	圆括号和方括号
*, +, ?, {n}, {n,}, {n,m}	限定符
^, \$, \ (元字符)	位置和顺序
	“或”运算

1.6.6 递归匹配

递归匹配在匹配具有嵌套结构的字符串时特别有效。给定算术表达式 $((1+2)*(3+4))$ ，该表达式具有嵌套结构。如果需要使用正则表达式检查该表达式的结构是否正确，则使用递归匹配能够解决该问题。

注意：本节下面介绍的递归匹配为 .NET Framework 所支持。

在 .NET Framework 中，正则表达式用于递归匹配的表达式说明如表 1.13 所示。

表 1.13 用于递归匹配的表达式说明

表达式	说明
(?<name>expression)	把匹配的内容命名为name，并压入堆栈
(?<-name>expression')	从堆栈中弹出最后压入的命名为name的匹配内容。如果堆栈为空，则当前组匹配失败
(?(name)yes no)	如果堆栈上存在命名为name的匹配内容，则继续匹配yes部分的表达式，否则继续匹配no部分的表达式
(?!)	零宽负向先行断言。由于没有后缀表达式，因此匹配总是失败

以下正则表达式能够匹配算术表达式 $((1+2)*(3+4))$ 。

```
\([^( )]*(((?<bracket>\([^( )]*+)((?<-bracket>\))[^( )]*+))*?(bracket)(?!))\)
```

(80)

下面是对该正则表达式进行的详细分析。

```
\(          # 匹配最外层的左括号
  [^( )]*  # 匹配最外层左括号后面的、不是括号“()”的内容
  (
    (
      (?<bracket>\( )  # 如果匹配到左括号，则命名为 bracket，并压入堆栈
      [^( )]*        # 匹配当前左括号后面的、不是括号“()”的内容
    )+
  )
```

```

(
    (?<-bracket>\() # 如果匹配到右括号, 则弹出命名为 bracket 的内容
    [^()]*          # 匹配右括号后面的, 不是括号 “()” 的内容
)+
)*
(?: (bracket) (?!)) # 如果匹配到最外层的右括号前面, 则检查堆栈内容是否为空。如果不为空,
                    # 则匹配失败
\)                # 匹配最外层的右括号

```

使用工具 **Regex Tester** 测试正则表达式 `\([^\)]*(((?<bracket>\([^\)]*+)((?<-bracket>\)[^\)]*+)*?(bracket)(?!))\)`, 结果如图 1.36 所示。

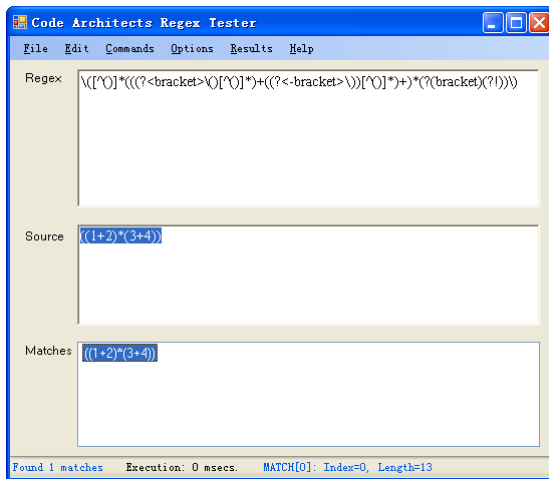


图 1.36 测试正则表达式 `\([^\)]*(((?<bracket>\([^\)]*+)((?<-bracket>\)[^\)]*+)*?(bracket)(?!))\)`

1.7 典型正则表达式解释

本节解释基于正则表达式基础理论的典型正则表达式。如匹配 Windows 运算系统的名称、匹配 HTML 标记、匹配 HTML 标记之间的内容。

1.7.1 匹配 Windows 运算系统的名称

Windows 运算系统存在很多版本, 如 Windows 95、Windows 98、Windows 2000、Windows ME、Windows XP、Windows 2003、Windows 7、Windows 8 等。以下正则表达式能够精确匹配 Windows 运算系统的名称。

```
Windows\s*((95)|(98)|(2000)|(2003)|(ME)|(XP)|(7)|(8))
```

(81)

上述表达式能够精确匹配 Windows 95、Windows 98、Windows 2000、Windows ME、Windows XP、Windows 2003、Windows 7、Windows 8 等运算系统的名称。然而, 精确匹配 Windows 运算系统名称的正则表达式比较冗长, 以下正则表达式能够简单匹配 Windows 运算系统的名称。

```
Windows\s\w+
```

(82)

1.7.2 匹配 HTML 标记

HTML 标记一般被尖括号包围, 如 `<a>`、`<table>`、`
`、`<input>` 等。以下正则表达式能够匹配 HTML 标记。

```
<[a-zA-Z][^>]*>
```

(83)

该正则表达式解释如下。

- ❑ <匹配 HTML 标记的左尖括号。
- ❑ 字符类[a-zA-Z]可以匹配一个英文字母，它匹配 HTML 标记中除去左尖括号的第一个字符。
- ❑ 字符类[^>]可以匹配除右尖括号之外的任何字符。
- ❑ [^>]*可以匹配空字符串，或者由除右尖括号之外的任何字符组成的字符串。
- ❑ >匹配 HTML 标记的右尖括号。
- ❑ [a-zA-Z][^>]*匹配 HTML 标记的名称。

使用工具 Regex Tester 测试正则表达式<[a-zA-Z][^>]*>，结果如图 1.37 所示。

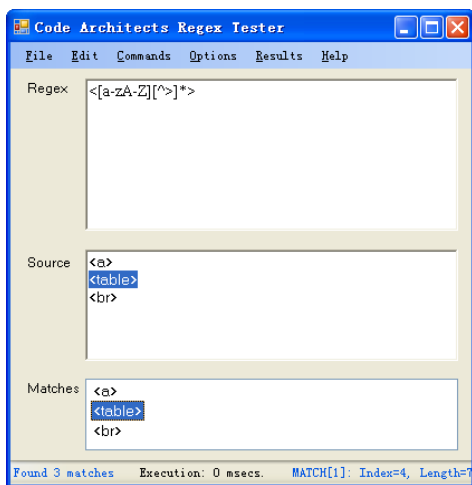


图 1.37 测试正则表达式<[a-zA-Z][^>]*>

1.7.3 匹配 HTML 标记之间的内容

如果要匹配 HTML 标记之间的内容，则可以使用以下代码：

```
(?<=<(?(tag)\w+)>).*?(?=<\/\k<tag>>)
```

(84)

该正则表达式解释如下。

- ❑ (?(tag)\w+)是一个分组，它的名称为“tag”。在该正则表达式匹配过程中，它将保存被匹配的内容。
- ❑ <(?(tag)\w+)>匹配 HTML 标记的开头标记，如<a>、<table>等。
- ❑ (?<=<(?(tag)\w+)>)是一个零宽度正回顾后发断言，它断言自身位置的前面能够匹配<(?(tag)\w+)>所匹配的内容。在此，该表达式断言被匹配的字符串的开头部分是 HTML 标记的开头标记。
- ❑ \k<tag>反向引用名称为“tag”的分组，即被匹配到的 HTML 标记的名称。
- ❑ \/匹配字符/。
- ❑ <\/\k<tag>>匹配 HTML 标记的结尾标记，如、</table>等。
- ❑ (?=<\/\k<tag>>)是一个零宽度正预测先行断言。在此，该表达式断言被匹配的字符串的结尾部分是 HTML 标记的结尾标记。
- ❑ .*匹配 HTML 标记之间的任何字符。

使用工具 Regex Tester 测试正则表达式`(?<=<(?(tag>\w+)>).*?(?=<\k<tag>>)`，结果如图 1.38 所示。



图 1.38 测试正则表达式`(?<=<(?(tag>\w+)>).*?(?=<\k<tag>>)`

1.7.4 匹配 CSV 文件内容

CSV 文件是非常特殊的一种文件，它的内容满足以下 4 个条件。

- 数据被逗号(,)分割。在此，设被分割后的数据称为单个数据。
- 单个数据的长度不能为 0。
- 如果某单个数据中包含逗号(,)，那么该单个内容被双引号(")包围。
- 如果单个数据中包含双引号(")，则用两个双引号表示一个双引号。

以下内容是某个 CSV 文件的一部分。

```
aaa,bbb,ccc,ddd,"ab,de",eee,fff,ggg,"this is ""aa",hhh
```

以下正则表达式能够匹配不包含逗号(,)或者双引号(")的单个数据。

```
[^",,]+ (85)
```

如果单个数据包含逗号(,)或者双引号(")时，那么该单个数据被双引号(")包围。因此，匹配该类型单个数据的正则表达式形式如下：

```
"[匹配单个数据内容的正则表达式]" (86)
```

该类型单个数据要么是两个连续双引号(""），要么是除单个双引号("")外的任意字符。

因此，以下正则表达式能够匹配包含逗号(,)或者双引号(")的单个数据。

```
"(?:([^\"]|"))+ (87)
```

综上所述，以下正则表达式能够匹配 CSV 文件中的单个数据。

```
[^",,]+|"(?:([^\"]|"))+ (88)
```

以下正则表达式能够匹配 CSV 文件的内容。

```
([^\",,]+|"(?:([^\"]|"))+")(\s*,\s*([^\",,]+|"(?:([^\"]|"))+"))* (89)
```

该正则表达式解释如下。

- `[^",,]+|"(?:([^\"]|"))+"`匹配 CSV 文件中的单个数据。
- `\s*,\s*`匹配逗号(,)两边的空白字符。
- `\s*,\s*(?:[^\",,]+|"(?:([^\"]|"))+")`匹配“逗号(,) + 单个数据”组成的字符串，并且还包含逗号(,)两边的空白字符。

- `(\s*,\s*([^\,]+|"(?:([^\"]|\\")+")"))*` 匹配 0 个或多个“逗号(,) + 单个数据”组成的字符串。同样，也包含逗号(,) 两边的空白字符。

使用工具 Regex Tester 测试正则表达式`([^\,]+|"(?:([^\"]|\\")+")")(\s*,\s*([^\,]+|"(?:([^\"]|\\")+")"))*`, 结果如图 1.39 所示。

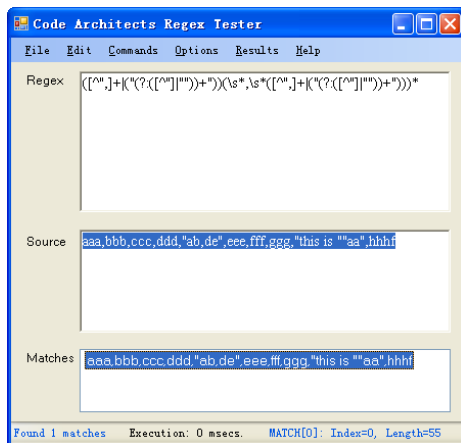


图 1.39 测试正则表达式`([^\,]+|\"(?:([^\"]|\\\")+\")*)(\s*,\s*([^\,]+|\"(?:([^\"]|\\\")+\")))*`

第 2 章 数字验证的方法

本章的内容主要介绍数字验证，以及与数字相关的常用应用字符串的验证方法。例如，数值验证、国内电话号码的验证、身份证验证、银行卡号和信用卡号验证、邮政编码验证和 IP 地址验证等。

在本章中所涉及的一些正则表达式中，有些使用元字符**b**来指定单词的边界，有些使用元字符**^**和**\$**来指定字符串的边界，有些则没有使用指定边界的元字符。读者在使用这些正则表达式时，可以根据自己的需要来确定是否使用以及所使用指定边界的元字符。下面是一些建议。

- 如果被验证的字符串是单词中的一部分，则可以不使用指定边界的元字符，或者仅仅使用指定开始位置（被验证的字符串在单词的开头）或结束位置（被验证的字符串在单词的结尾）的元字符**b**。
- 如果被验证的字符串是一个完整的单词，则可以使用元字符**b**来指定单词的开始和结束位置。
- 如果被验证的字符串是一个完整的字符串，则可以使用元字符**^**和**\$**来分别指定字符串的开始和结束位置。

2.1 9 种数值验证

本节介绍数值的验证方法，主要包括数字验证、整数验证、指定范围的整数验证、实数验证、指定精度的实数验证、科学计数法的数值验证、二进制数值验证、八进制数值验证和十六进制数值验证。

2.1.1 字符串只包含数字的验证

本节讲解的字符串只包含数字，即它只包含 0、1、2、3、4、5、6、7、8、9 数字字符。要验证字符串是否只包含数字，可以使用元字符**d**、字符类**[0-9]**或者字符类**[0123456789]**。下面介绍验证只包含数字的字符串的方法。

1. 只包含数字的任意长度的字符串验证

只包含数字任意长度的字符串，即它的长度可以为 0、1 或者大于 1 的整数。以下正则表达式都能够验证只包含数字、长度大于或等于 0 的字符串。

<code>\d*</code>	(1)
<code>\d{0,}</code>	(2)
<code>[0-9]*</code>	(3)
<code>[0123456789]*</code>	(4)

以下正则表达式能够验证只包含数字、长度大于或等于 1 的字符串。

<code>\d+</code>	(5)
<code>\d{1,}</code>	(6)
<code>\d\d*</code>	(7)

<code>\d?\d+</code>	(8)
<code>\d{0,1}\d{1,}</code>	(9)
<code>[0-9]+</code>	(10)
<code>[0123456789]+</code>	(11)

正则表达式解释如下。

- 正则表达式`\d?\d+`使用了两个限定符：`?`和`+`。`\d?`可以匹配零个或一个数字；`\d+`可以匹配长度至少为 1 的数字字符串。给定数字字符串“012345”，`\d?`首先匹配第一个数字 0，`\d+`匹配从第二个数字开始的后面所有数字。
- 在正则表达式`\d{0,1}\d{1,}`中，`\d{0,1}`可以匹配零个或一个数字；`\d{1,}`可以匹配长度至少为 1 的数字字符串。给定数字字符串“012345”，`\d{0,1}`首先匹配第一个数字 0，`\d{1,}`匹配从第二个数字 1 开始的后面所有数字。

使用工具 Regex Tester 分别测试正则表达式`\d?\d+`和`\d{0,1}\d{1,}`，结果分别如图 2.1 和图 2.2 所示。

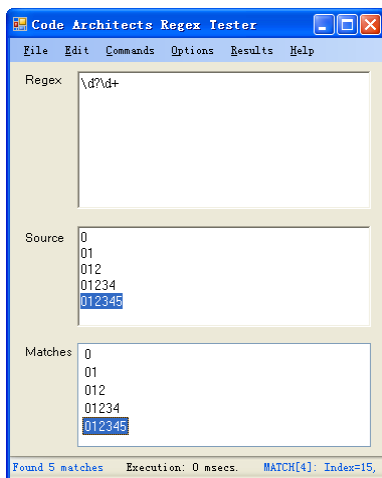


图 2.1 测试正则表达式`\d?\d+`

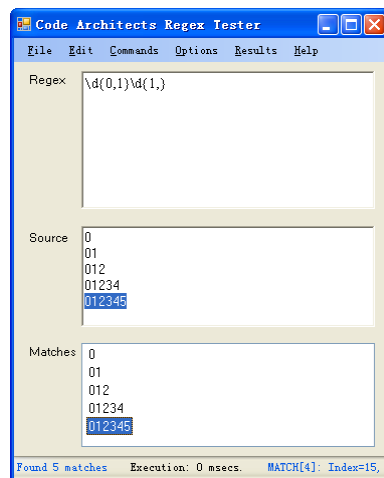


图 2.2 测试正则表达式`\d{0,1}\d{1,}`

2. 只包含数字的指定长度的字符串

只包含数字的、指定长度的字符串的长度是固定的，不妨设字符串的长度为 N （其中， $N \geq 1$ ）。最简单的情况就是 N 等于 1，即验证一位整数。此时，只需要一个`\d`或`[0-9]`表达式就能够验证一位整数。以下正则表达式都能够验证只包含数字、长度等于 1 的字符串。

<code>\d</code>	(12)
<code>\d{1,1}</code>	(13)
<code>[0-9]</code>	(14)
<code>[0123456789]</code>	(15)

若 N 是一个比较小的整数，则可以使用 N 个`\d`或`[0-9]`表达式来验证长度为 N 、只包含数字的字符串。不妨设 N 等于 3，则以下两个正则表达式都能够验证只包含数字、长度等于 3 的字符串。

<code>\d\d\d</code>	(16)
<code>[0-9][0-9][0-9]</code>	(17)

若 N 是一个比较大的整数，则可以使用限定符来验证只包含数字、指定长度的字符串。以下几个正则表达式都能够验证只包含数字、长度等于 N 的字符串。

<code>\d{N}</code>	(18)
--------------------	------

$\backslash d\{N,N\}$	(19)
$[0-9]\{N,N\}$	(20)
$\backslash d\backslash d\{N-1,N-1\}$	(21)
$[0-9][0-9]\{N-1,N-1\}$	(22)

正则表达式解释如下。

- 正则表达式 $\backslash d\{N,N\}$ 可以匹配长度为 N 的字符串。
- 在正则表达式 $\backslash d\backslash d\{N-1,N-1\}$ 中， $\backslash d$ 可以匹配一个数字； $d\{N-1,N-1\}$ 可以匹配长度为 $N-1$ 的数字字符串。不妨设 N 等于 6，给定数字字符串“012345”， $\backslash d$ 首先匹配第一个数字 0， $d\{5,5\}$ 匹配从第二个数字 1 开始的后面所有数字。

不妨设 N 等于 6，使用工具 Regex Tester 分别测试正则表达式 $\backslash d\{6,6\}$ 和 $\backslash d\backslash d\{5,5\}$ ，结果分别如图 2.3 和图 2.4 所示。

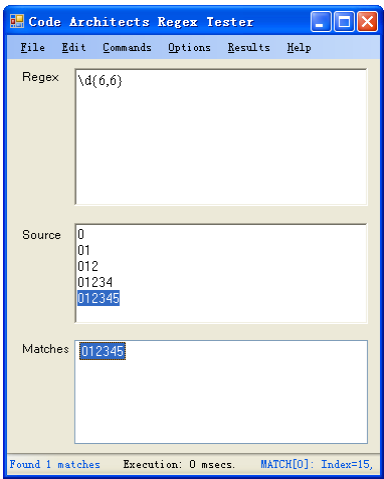


图 2.3 测试正则表达式 $\backslash d\{6,6\}$

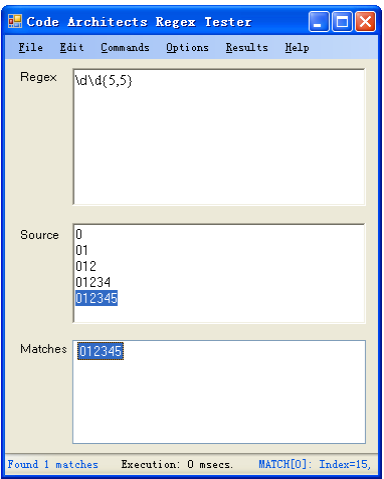


图 2.4 测试正则表达式 $\backslash d\backslash d\{5,5\}$

3. 只包含数字的指定范围长度的字符串

只包含数字、指定范围长度的字符串，即字符串的长度在一个指定的范围内。不妨设字符串的长度为 L ， $N \leq L \leq M$ (N 和 M 为整数，且 $M \geq N$)。验证该类型字符串的最简单的方法就是使用限定符 $\{N,M\}$ 。以下正则表达式都能够验证只包含数字、最小长度为 N 、最大长度为 M 的字符串。

$\backslash d\{N,M\}$	(23)
$[0-9]\{N,M\}$	(24)
$\backslash d\backslash d\{N-1,M-1\}$ # N, M 均大于 0	(25)

正则表达式解释如下。

- 正则表达式 $\backslash d\{N,M\}$ 可以匹配最小长度为 N 、最大长度为 M 的数字字符串。
- 在正则表达式 $\backslash d\backslash d\{N-1,M-1\}$ 中， $\backslash d$ 可以匹配一个数字， $d\{N-1,M-1\}$ (其中， N, M 大于 0) 可以匹配最小长度为 $N-1$ 、最大长度 $M-1$ 的数字字符串。不妨设 N 等于 4、 M 等于 6，给定数字字符串“012345”， $\backslash d$ 首先匹配第一个数字 0， $d\{3,5\}$ 匹配从第二个数字 1 开始的后面长度为 3、4 或者 5 的数字字符串。

不妨设 N 等于 4、 M 等于 6，使用工具 Regex Tester 分别测试正则表达式 $\backslash d\{4,6\}$ 和 $\backslash d\backslash d\{3,5\}$ ，结果分别如图 2.5 和图 2.6 所示。

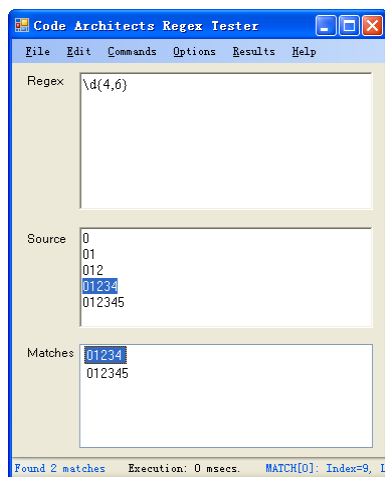


图 2.5 测试正则表达式\d{4,6}

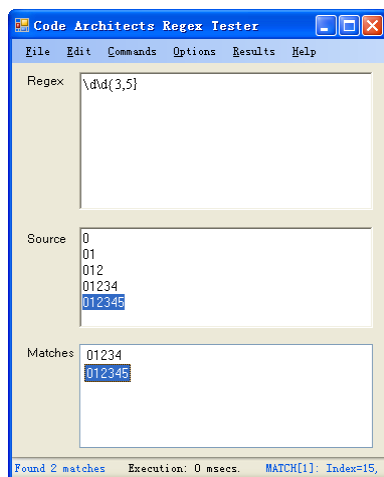


图 2.6 测试正则表达式\d\d{3,5}

4. 只包含数字的指定范围长度的单词字符串

只包含数字、指定范围长度的单词字符串，即字符串的长度在一个指定的范围内，且该字符串具有开始和结束边界。不妨设字符串的长度为 L ， $N \leq L \leq M$ (N 和 M 为整数，且 $M \geq N$)。验证该类型字符串的最简单的方法就是使用限定符 $\{N,M\}$ 。以下正则表达式都能够验证只包含数字、最小长度为 N 、最大长度为 M 的单词字符串。

$\backslash b \backslash d \{N,M\} \backslash b$ (26)

$\backslash b [0-9] \{N,M\} \backslash b$ (27)

正则表达式解释如下。

- 在正则表达式 $\backslash b \backslash d \{N,M\} \backslash b$ 中， $\backslash b$ 匹配单词的边界，即单词的开始位置或结束位置； $\backslash d \{N,M\}$ 可以匹配最小长度为 N 、最大长度为 M 的数字字符串。不妨设 N 等于 4、 M 等于 6，给定数字字符串“012345”。 $\backslash b$ 首先匹配单词的开始位置， $\backslash d \{4,6\}$ 匹配从第一个数字开始的长度为 4、5 或者 6 的数字字符串，最后， $\backslash b$ 匹配单词的结束位置。

使用工具 Regex Tester 测试正则表达式 $\backslash b \backslash d \{4,6\} \backslash b$ ，结果如图 2.7 所示。

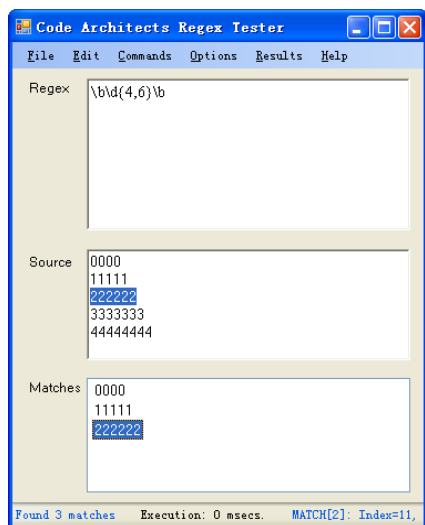


图 2.7 测试正则表达式\b\d{4,6}\b

2.1.2 字符串只包含整数的验证

本节讲解的字符串为整数。它只能包含 0、1、2、3、4、5、6、7、8、9 数字字符，并且第一个数字不能为 0（即除整数 0 之外的数字）。要验证整数字符串，可以使用元字符 `\d`、字符类 `[0-9]` 或者字符类 `[0123456789]`。下面介绍验证整数字符串的方法。

注意：2.1.1 小节中介绍的只包含数字的字符串不一定是整数，如“012345”。

1. 非负整数

非负整数是大于或者等于 0 的整数。它的验证可以分为两种情况：0 的验证和大于 0 的整数的验证。验证大于 0 的整数时，需要验证整数的第一位（最左边的数字）不能为数字 0。以下正则表达式都能够验证非负整数。

```
\b(0|[1-9]\d*)\b (28)
```

```
\b(0|[1-9]\d{0,})\b (29)
```

```
\b(0|[1-9][0-9]*)\b (30)
```

正则表达式解释如下。

- 在正则表达式 `\b(0|[1-9]\d*)\b` 中，`\b` 匹配整数的边界；`0` 匹配整数 0；`[1-9]` 匹配 1~9 中的任意数字，它将匹配整数的最高位（最左边的数字）；`\d*` 匹配整数，但不包括最高位之外的部分（如果存在）。

2. 负整数

负整数是小于 0 的整数（必须以字符“-”开头），它的验证和正整数的验证相似。以下正则表达式都能够验证负整数。

```
-[1-9]\d*\b (31)
```

```
-[1-9]\d{0,}\b (32)
```

```
-[1-9][0-9]*\b (33)
```

正则表达式解释如下。

- 在正则表达式 `-[1-9]\d*\b` 中，`\b` 匹配整数的边界；`-` 匹配负整数的字符“-”；`[1-9]` 匹配 1~9 中的任意数字，它将匹配整数最左边的数字；`\d*` 匹配整数，但不包括最左边数字之外的部分（如果存在）。

3. 任意整数

整数包括正整数、0 和负整数。把上述两种验证综合起来，就可以验证任意整数。因为，整数分为正整数、0 和负整数，因此需要按照这三种情况来验证。以下正则表达式都能够验证任意整数。

```
^(0|-?[1-9]\d*)\b (34)
```

```
^(0|-?[1-9]\d{0,})\b (35)
```

```
^(0|-?[1-9][0-9]*)\b (36)
```

正则表达式解释如下。

- 正则表达式 `(0|-?[1-9]\d*)\b`。`\b` 匹配整数的边界；`-?` 表示字符“-”可以不出现或者出现一次，如果不出现，则匹配正整数，否则匹配负整数；`[1-9]` 匹配 1~9 中的任意数字，它将匹配整数最左边的数字；`\d*` 匹配整数，但不包括最左边数字之外的部分（如果存在）。
- 正则表达式 `(0|-?[1-9][0-9]*)\b`。`\b` 匹配整数的边界；`-?` 表示字符“-”可以不出现或者出现一次，如果不出现，则匹配正整数，否则匹配负整数；`[1-9]` 匹配 1~9 中的任意数字，它将匹配整数最左边的数字；`[1-9]*` 匹配整数，但不包括最左边数字之外的部分（如果存在）。

使用工具 Regex Tester 分别测试正则表达式 $(0|-?[1-9]\text{d}^*)\text{b}$ 和 $(0|-?[1-9][0-9]^*)\text{b}$ ，结果分别如图 2.8 和图 2.9 所示。

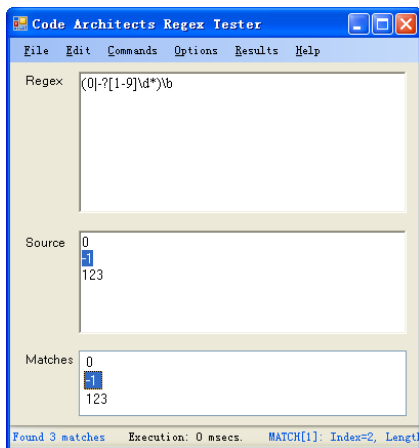


图 2.8 测试正则表达式 $(0|-?[1-9]\text{d}^*)\text{b}$

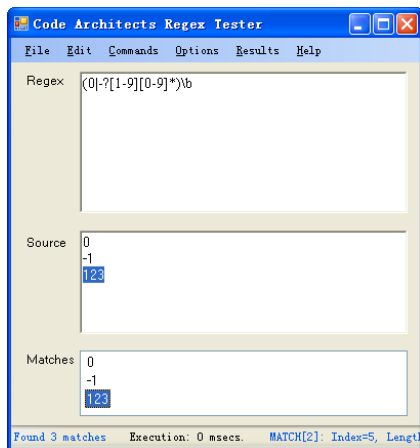


图 2.9 测试正则表达式 $(0|-?[1-9][0-9]^*)\text{b}$

2.1.3 字符串指定范围的整数验证

本节讲解的字符串为指定范围的整数。在此，本节将介绍指定位数的整数和指定值范围的整数的验证。其中，指定位数的整数又包括指定位数的正整数和负整数；指定值范围的整数又包括指定值范围的正整数和负整数。

1. 指定位数的正整数

指定位数的正整数，即它的位数是固定的，不妨设为 N (N 为大于 0 的整数)。最简单的情况就是 N 等于 1，即验证一位正整数（包括 0）。此时，只需要一个 d 或 $[0-9]$ 表达式就能够验证一位正整数（包括 0）。以下正则表达式都能够验证一位正整数（包括 0）。

```
\b\d\b (37)
\b\d{1,1}\b (38)
\b[0-9]\b (39)
\b[0123456789]\b (40)
```

若 N 大于或等于 1 时，该正整数的验证和一位正整数的验证存在一些区别。此时，该正整数的第一位不能为 0。不妨设 N 等于 3，以下正则表达式都能够验证 3 位正整数。

```
\b[1-9]\d{2}\b (41)
\b[1-9]\d\d\b (42)
\b[1-9][0-9][0-9]\b (43)
\b1\d{2}|2\d{2}|3\d{2}|4\d{2}|5\d{2}|6\d{2}|7\d{2}|8\d{2}|9\d{2}\b (44)
\b(1|2|3|4|5|6|7|8|9)\d{2}\b (45)
```

正则表达式解释如下。

- 正则表达式 $\text{b}[1-9]\text{d}\{2\}\text{b}$ 。 b 匹配单词的边界； $[1-9]$ 匹配 1~9 中的任意数字，它匹配正整数的第一位（即百分位）； $\text{d}\{2\}$ 匹配正整数的后面两位。不妨设被匹配的正整数为 145，那么 b 匹配整数的边界； $[1-9]$ 匹配百位数字 1； $\text{d}\{2\}$ 匹配整数的后面两位 45。
- 正则表达式 $\text{b}(1|2|3|4|5|6|7|8|9)\text{d}\{2\}\text{b}$ 。 b 匹配单词的边界； $(1|2|3|4|5|6|7|8|9)$ 匹配 1~9 中的任意数字，它匹配正整数的第一位（即百分位）； $\text{d}\{2\}$ 匹配正整数的后面两位。不妨设被匹配的正整数为 145，那么 b 匹配正整数的边界； $(1|2|3|4|5|6|7|8|9)$ 匹配百位 1； $\text{d}\{2\}$

匹配正整数的后面两位 45。

使用工具 Regex Tester 分别测试正则表达式 `\b[1-9]\d{2}\b` 和 `\b(1|2|3|4|5|6|7|8|9)\d{2}\b`，结果分别如图 2.10 和 2.11 所示。

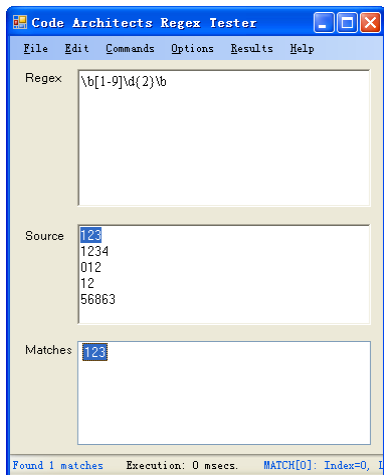


图 2.10 测试正则表达式 `\b[1-9]\d{2}\b`

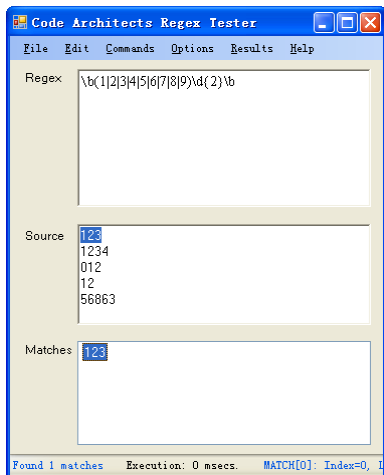


图 2.11 测试正则表达式 `\b(1|2|3|4|5|6|7|8|9)\d{2}\b`

2. 指定位数的负整数

指定位数的负整数，即它的位数是固定的，不妨设为 N (N 为大于 0 的整数)。不妨设 N 等于 2。2 位负整数比 2 位正整数多一个字符“-”。除了字符“-”外，其他部分的验证方式和 2 位正整数的验证方式相同。以下正则表达式都能够验证 2 位负整数。

<code>-[1-9]\d\b</code>	(46)
<code>-[1-9]\d{1}\b</code>	(47)
<code>-[1-9][0-9]\b</code>	(48)
<code>-1\d 2\d 3\d 4\d 5\d 6\d 7\d 8\d 9\d\b</code>	(49)
<code>-(1 2 3 4 5 6 7 8 9)\d\b</code>	(50)

正则表达式解释如下。

- 正则表达式 `-[1-9]\d\b`。`\b` 匹配单词的边界；`-` 匹配字符“-”；`[1-9]` 匹配 1~9 中的任意数字，它匹配负整数最左边的数字；`\d` 匹配负整数除左边第一位数字之外后面的数字。不妨设给定负整数-98：`\b` 匹配整数的边界；`-` 匹配字符“-”；`[1-9]` 匹配最左边的数字 9；`\d` 匹配负整数除左边第一位数字之外后面的数字 8。
- 正则表达式 `-(1|2|3|4|5|6|7|8|9)\d\b`。`\b` 匹配单词的边界；`-` 匹配字符“-”；`[1-9]` 匹配 1~9 中的任意数字，它匹配负整数最左边的数字；`\d` 匹配负整数除左边第一位数字之外后面的数字。不妨设给定负整数-56：`\b` 匹配整数的边界；`-` 匹配字符“-”；`[1-9]` 匹配最左边的数字 5；`\d` 匹配负整数除左边第一位数字之外后面的数字 6。

3. 指定范围的正整数

指定范围的正整数，即它的数值在一定的范围（必须是正整数）之内。在此，不妨设该范围为 1~5678。为了验证 1~5678 范围内的整数，在此，将该范围划分为以下 5 个范围。

- 1~999
- 1000~4999
- 5000~5599

❑ 5600~5669

❑ 5670~5678

以下正则表达式能够验证 1~999 范围的正整数。

```
\b[1-9]\d{0,2}\b (51)
```

以下正则表达式能够验证 1000~4999 范围的正整数。

```
\b[1-4]\d{3}\b (52)
```

以下正则表达式能够验证 5000~5599 范围的正整数。

```
\b5[0-5]\d{2}\b (53)
```

以下正则表达式能够验证 5600~5669 范围的正整数。

```
\b56[0-6]\d\b (54)
```

以下正则表达式能够验证 5670~5678 范围的正整数。

```
\b567[0-8]\b (55)
```

综合以上，可以得到验证 1~5678 范围内的整数的正则表达式，具体如下：

```
\b(567[0-8])|(56[0-6]\d)|(5[0-5]\d{2})|([1-4]\d{3})|([1-9]\d{0,2})\b (56)
```

注意：在组合上述 5 个范围各自的正则表达式时，组合的顺序是固定的，即必须按照(5)、(4)、(3)、(2)、(1)的顺序组合。否则，其他顺序组合的表达式有可能不能验证某一个范围内的整数。若(1)表达式组合在(2)表达式之前，即组合为正则表达式**b([1-9]\d{0,2})|([1-4]\d{3})\b**，将不能验证 1000~4999 范围内的整数。原因如下：如果被验证的整数为 1234，则正则表达式**b([1-9]\d{0,2})|([1-4]\d{3})\b**将首先匹配整数 1234 中的 234 部分，并结束这个匹配过程。此时，该表达式永远不会匹配整数 1234。

组合顺序原则（正整数）：依次从最大值的范围组合到最小值的范围。

使用工具 Regex Tester 测试正则表达式**\b(567[0-8])|(56[0-6]\d)|(5[0-5]\d{2})|([1-4]\d{3})|([1-9]\d{0,2})\b**，结果如图 2.12 所示。

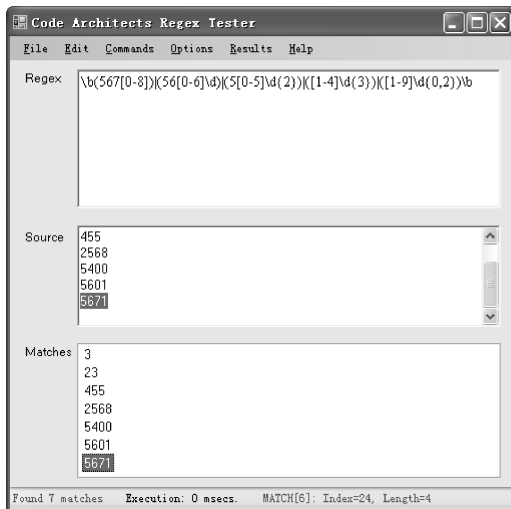


图 2.12 测试正则表达式**\b(567[0-8])|(56[0-6]\d)|(5[0-5]\d{2})|([1-4]\d{3})|([1-9]\d{0,2})\b**

4. 指定范围的负整数

指定范围的负整数，即它的数值在一定的范围（必须是负整数）之内。在此，不妨设该范围

为-128~-25。为了验证-128~-25 范围内的整数，把该范围划分为以下 4 个范围。

- ❑ -128~-120
- ❑ -119~-100
- ❑ -99~-30
- ❑ -29~-25

以下正则表达式能够验证-128~-120 范围的正整数。

```
-12[0-8]\b (57)
```

以下正则表达式能够验证-119~-100 范围的正整数。

```
-1[0-1]\d\b (58)
```

以下正则表达式能够验证-99~-30 范围的正整数。

```
-[3-9]\d\b (59)
```

以下正则表达式能够验证-29~-25 范围的正整数。

```
-2[5-9]\b (60)
```

综合以上，可以得到验证-128~-25 范围内的整数的正则表达式，具体如下：

```
-((12[0-8])|(1[0-1]\d)|([3-9]\d)|(2[5-9]))\b (61)
```

注意：在组合上述 4 个范围各自的正则表达式时，组合的顺序是固定的，即必须按照(57)、(58)、(59)、(60)的顺序组合。

组合顺序原则（正整数）：依次从最小值的范围组合到最大值的范围。

组合顺序原则（负整数）：依次从绝对值最大值的范围组合到绝对值最小值的范围。

使用工具 Regex Tester 测试正则表达式-((12[0-8])|(1[0-1]\d)|([3-9]\d)|(2[5-9]))\b，结果如图 2.13 所示。

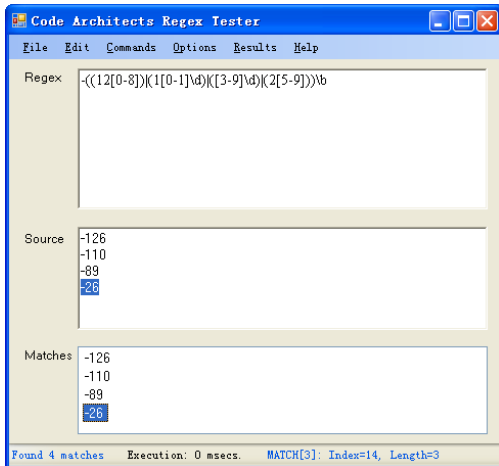


图 2.13 测试正则表达式-((12[0-8])|(1[0-1]\d)|([3-9]\d)|(2[5-9]))\b

2.1.4 字符串为实数的验证

本节讲解的字符串为实数的验证方法。为了使得介绍的内容不与整数范围的验证重复，在此，本节将介绍被验证的实数至少包含一个小数点（.）。

实数由 3 部分组成：整数部分、小数部分和小数点。其中，小数点分割整数部分和小数部分。

1. 实数 0

实数 0 的表示方法可以是 0.0、0.00、0.000……其中，小数点后面的 0 的数量至少为 1。以下正则表达式都能够验证实数 0。

```
\b0\.0+\b (62)
```

```
\b0\.0{1,}\b (63)
```

```
\b0\.00*\b (64)
```

正则表达式解释如下。

- ❑ 正则表达式 `\b0\.0+\b`。 `\b` 匹配整数的边界；第一个 0 匹配小数点前面的 0；`\.` 使用转义字符 `\` 进行转义，它匹配小数点；`0+` 匹配小数点后面所有的 0（至少为 1 个）。
- ❑ 正则表达式 `\b0\.00*\b`。 `\b` 匹配整数的边界；第一个 0 匹配小数点前面的 0；`\.` 使用转义字符 `\` 进行转义，它匹配小数点；第二个 0 匹配小数点后面第一个 0；`0*` 匹配从小数点后面第二个 0 开始的所有的 0（如果存在）。

2. 正实数

正实数是大于 0 的实数。它的验证可以分为 3 部分：验证整数部分、验证小数点和验证小数部分。其中，整数部分验证包括两种情况：0 和正整数的验证。验证整数部分其实就是验证一个非负整数，以下正则表达式都能够验证正实数的整数部分。

```
0|([1-9]\d*) (65)
```

```
0|([1-9]\d{0,}) (66)
```

```
0|([1-9][0-9]*) (67)
```

验证小数点部分比较简单，以下正则表达式能够验证小数点部分。

```
\. (68)
```

验证小数部分比验证正整数稍微复杂。小数部分和正整数的差别如下。

- ❑ 小数部分的第一位（最左边的数字）可以是 0，而正整数不可以。
- ❑ 小数部分可以添加无限多个 0，即 123.45600、123.456000 和 123.456 的值相等。

以下正则表达式都能够验证实数的小数部分。

```
\d+ (69)
```

```
\d\d* (70)
```

```
[0-9][0-9]* (71)
```

综合以上，可以得到验证正实数的正则表达式如下：

```
(0|([1-9]\d*))\.\d+ (72)
```

使用工具 Regex Tester 测试正则表达式 `(0|([1-9]\d*))\.\d+`，结果如图 2.14 所示。

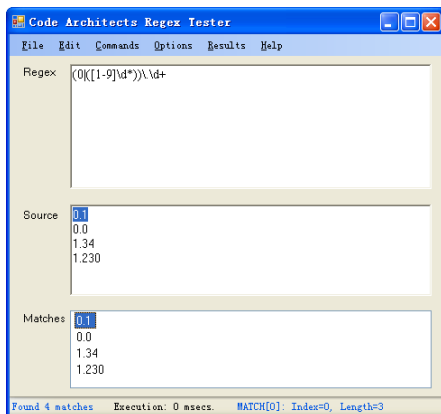


图 2.14 测试正则表达式 `(0|([1-9]\d*))\.\d+`

3. 负实数

负实数是小于 0 的实数。它的验证可以分为三部分：验证整数部分、验证小数点和验证小数部分。其中，整数部分验证只包括负整数（包括负 0）的验证。因此，验证整数部分其实就是验证一个负整数，以下正则表达式都能够验证负实数的整数部分。

```
-(0|([1-9]\d*)) (73)
```

```
-(0|([1-9]\d{0,})) (74)
```

```
-(0|([1-9][0-9]*)) (75)
```

验证小数点部分比较简单，以下正则表达式能够验证小数点部分。

```
\. (76)
```

验证小数部分比验证正整数稍微复杂。小数部分和正整数的差别如下。

- 小数部分的第一位（最左边的数字）可以是 0，而正整数却不可以。
- 小数部分可以添加无限多个 0，即 123.45600、123.456000 和 123.456 的值相等。

以下正则表达式都能够验证实数的小数部分。

```
\d+ (77)
```

```
\d\d* (78)
```

```
[0-9][0-9]* (79)
```

综合以上，可以得到验证负实数的正则表达式，具体如下：

```
-(0|([1-9]\d*))\.\d+ (80)
```

使用工具 Regex Tester 测试正则表达式`-(0|([1-9]\d*))\.\d+`，结果如图 2.15 所示。

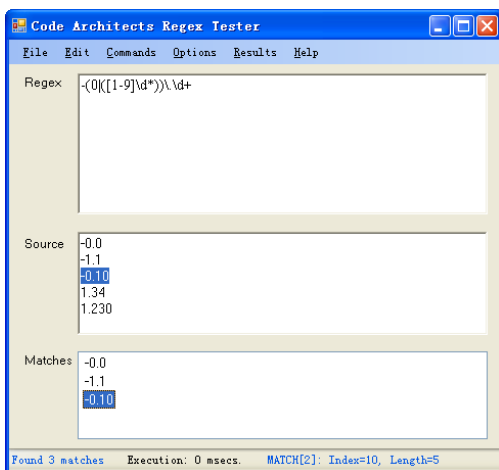


图 2.15 测试正则表达式`-(0|([1-9]\d*))\.\d+`

4. 实数

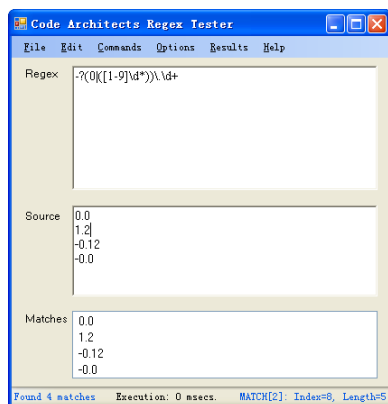
综合以上三点可知，如果要验证实数，只要把以上 3 种类型的正则表达式综合起来即可。以下正则表达式都能够验证实数。

```
-?(0|([1-9]\d*))\.\d+ (81)
```

```
-?(0|([1-9]\d{0,}))\.\d{1,} (82)
```

注意：正则表达式`-?(0|([1-9]\d*))\.\d+`除了能够验证一般形式的实数（如 0.0、1.2、-1.20 等）外，还能够验证负 0，即 -0.0、-0.00 等。

使用工具 Regex Tester 测试正则表达式`-?(0|([1-9]\d*))\.\d+`，结果如图 2.16 所示。

图 2.16 测试正则表达式`^-(0|([1-9]d*))\d+`

2.1.5 字符串指定精度的实数验证

本节讲解的字符串为指定精度的实数。它包括指定精度的正实数和负实数。指定精度实数的小数部分的位数是固定的，不妨设为 N （其中， N 为大于 0 的整数）。其中，指定精度的实数包括正实数和负实数。

在很多计数中，为了满足某种精度，往往要求数值的小数部分的位数是固定的。不妨设小数部分的位数为 N （其中， N 为大于 0 的整数）。以下正则表达式都能够验证小数部分的位数为 N 的实数。

`^-(0|([1-9]d*))\.\d{N}$` (83)

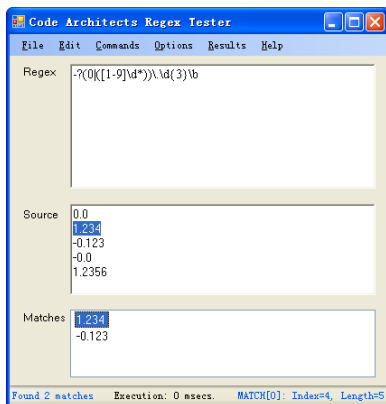
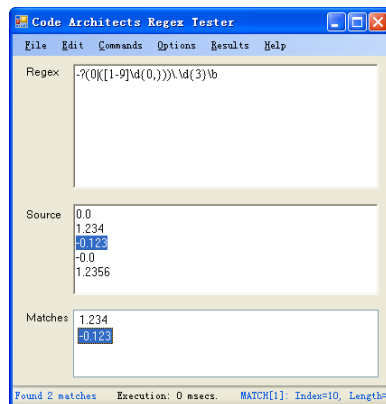
`^-(0|([1-9]d{0,}))\.\d{N}$` (84)

`^-(0|([1-9][0-9]*)\.\d{N}$` (85)

正则表达式解释如下。

- ❑ 正则表达式`^-(0|([1-9]d*))\.\d{N}$`。`^`、`$`分别匹配字符串的开始和结束位置；`0|([1-9]d*)`中的 `0` 匹配整数部分的数字 `0`，`([1-9]d*)` 匹配以非 `0` 开头的任意整数；`\.` 匹配小数点；`\d{N}` 匹配长度为 N 的小数部分。
- ❑ 正则表达式`^-(0|([1-9]d{0,}))\.\d{N}$`。`^`和`$`分别匹配字符串的开始和结束位置；`0|([1-9]d{0,})`中的 `0` 匹配整数部分的数字 `0`，`([1-9]d{0,})` 匹配以非 `0` 开头的任意整数；`\.` 匹配小数点；`\d{N}` 匹配长度为 N 的小数部分。

不妨设 N 等于 3，使用工具 `Regex Tester` 分别测试正则表达式`^-(0|([1-9]d*))\.\d{3}$`和`^-(0|([1-9]d{0,}))\.\d{3}$`，结果分别如图 2.17 和图 2.18 所示。

图 2.17 测试正则表达式`^-(0|([1-9]d*))\.\d{3}$`图 2.18 测试正则表达式`^-(0|([1-9]d{0,}))\.\d{3}$`

2.1.6 科学计数法的数值验证

科学计数法就是把一个数记成 $a \times 10^n$ 的形式。其中, a 是一位整数, 或者是只有一位整数的小数, 即 a 满足: $1 \leq |a| < 10$ 。 n 为一个整数。要验证科学计数法的数值, 其实只需验证数值 a 和 n , 除 a 和 n 之外的字符都是常量。

数值 a 可以为整数, 也可以为小数, 且 a 的整数位只能为 1~9。以下正则表达式都能够验证数值 a 。

```
^-?[1-9](\.\d+)?$ (86)
```

```
^-?[1-9](\.\d\d*)?$ (87)
```

以下正则表达式都能够验证科学计数法的数值 ($a \times 10^n$)。

```
^-?[1-9](\.\d+)?\*10\^-?\d+$ (88)
```

```
^-?[1-9](\.\d\d*)?\*10\^-?\d+$ (89)
```

正则表达式解释如下。

□ 正则表达式 `^-?[1-9](\.\d+)?$`。 `^` 和 `$` 分别匹配字符串的开始和结束位置; `-?` 表示字符 `-` 可以出现 0 次或者 1 次, 如果出现 0 次, 则匹配正数, 否则匹配负数; `[1-9]` 匹配数值 a 的整数部分; `(\.\d+)?` 表示 `\.\d+` 的匹配项可以出现 0 次或者 1 次, 如果出现 0 次, 则匹配整数, 否则匹配小数; `\.` 匹配字符 `.`; `\d+` 匹配数值 a 的小数部分。

□ 正则表达式 `^-?[1-9](\.\d+)*10\^-?\d+$`。 `^` 和 `$` 分别匹配字符串的开始和结束位置; `^-?[1-9](\.\d+)?` 匹配数值 a ; `*` 匹配字符 `*`; `10` 匹配数字 10; `\` 匹配字符 `\`; `-?\d+` 匹配整数 n 。

使用工具 Regex Tester 测试正则表达式 `^-?[1-9](\.\d+)*10\^-?\d+$`, 结果如图 2.19 所示。

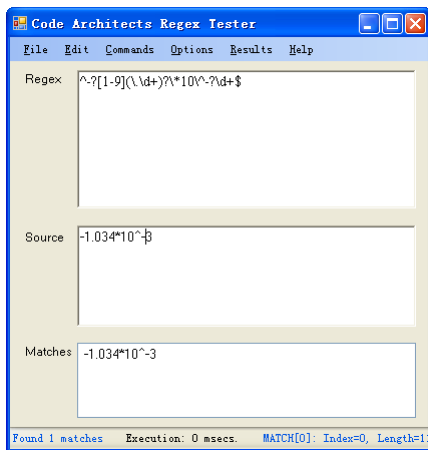


图 2.19 测试正则表达式 `^-?[1-9](\.\d+)*10\^-?\d+$`

2.1.7 二进制数值的验证

二进制数值只能包含 0 或者 1 两个数字, 且第一位不能为 0 (除 0 本身之外)。它的验证方法比较简单, 以下正则表达式都能够验证二进制数值。

```
^(0|(1[0-1]*))$ (90)
```

```
^(0|(1[0-1]{0,}))$ (91)
```

验证给定范围的二进制数值的方法比上述验证方法稍微复杂。在此, 不妨设二进制的范围为 1000~10000001。为了验证该范围内的二进制数值, 特将该范围划分为以下 3 个范围。

□ 1000~1111111

❑ 10000000

❑ 10000001

以下正则表达式能够验证 1000~1111111 范围的二进制数值。

```
^1[01]{3,6}$ (92)
```

以下正则表达式能够验证 10000000~10000001 范围的二进制数值。

```
^10{6}[01]$ (93)
```

综合以上，可以得到验证 1000~10000001 范围内二进制数值的正则表达式，具体如下：

```
^1(( [01]{3,6})|(0{6}[01]))$ (94)
```

正则表达式解释如下。

- ❑ 正则表达式`^1[01]{3,6}$`。`^`和`$`分别匹配字符串的开始和结束位置；第一个 1 匹配二进制数值最左边第一位数字 1；`[01]{3,6}`可以匹配由 0 或者 1 组成、最小长度为 3、最大长度为 6 的字符串。
- ❑ 正则表达式`^10{6}[01]$`。`^`和`$`分别匹配字符串的开始和结束位置；第一个 1 匹配二进制数值最左边第一位数字 1；`0{6}`匹配 6 个数字 0；`[01]`匹配二进制数值的最后一位，它可以是 0 或者 1。

使用工具 Regex Tester 测试正则表达式`^1((([01]{3,6}))(0{6}[01]))$`，结果如图 2.20 所示。

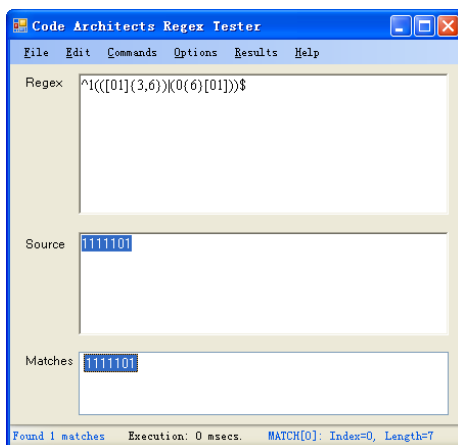


图 2.20 测试正则表达式`^1((([01]{3,6}))(0{6}[01]))$`

2.1.8 八进制数值的验证

八进制数值可以包含 0、1、2、3、4、5、6、7 这 8 个数字，且第一位不能为 0（除 0 本身之外）。它的验证方法比较简单，以下正则表达式都能够验证八进制数值。

```
^(0|([1-7][0-7]*))$ (95)
```

```
^(0|([1-7][0-7]{0,}))$ (96)
```

验证给定范围的八进制数值的方法比上述验证方法稍微复杂。在此，不妨设八进制的范围为 1~4567。为了验证该范围内的八进制数值，特将该范围划分为以下 5 个范围。

- ❑ 1-777
- ❑ 1000~3777
- ❑ 4000~4477
- ❑ 4500~4557
- ❑ 4560~4567

以下正则表达式能够验证 1~777 范围的八进制数值。

```
^[1-7][0-7]{0,2}$ (97)
```

以下正则表达式能够验证 1000~3777 范围的八进制数值。

```
^[1-3][0-7]{3}$ (98)
```

以下正则表达式能够验证 4000~4477 范围的八进制数值。

```
^4[0-4][0-7]{2}$ (99)
```

以下正则表达式能够验证 4500~4557 范围的八进制数值。

```
^45[0-5][0-7]$ (100)
```

以下正则表达式能够验证 4560~4567 范围的八进制数值。

```
^456[0-7]$ (101)
```

综合以上，可以得到验证 1~4567 范围的八进制数值的正则表达式，具体如下：

```
^((456[0-7])|(45[0-5][0-7])|(4[0-4][0-7]{2})|([1-3][0-7]{0,3})|([1-7][0-7]{0,2}))$ (102)
```

正则表达式解释如下。

- 正则表达式`^((456[0-7])|(45[0-5][0-7])|(4[0-4][0-7]{2})|([1-3][0-7]{0,3})|([1-7][0-7]{0,2}))$`。^和\$分别匹配字符串的开始和结束位置；456[0-7]验证 4560~4567 范围的八进制数值；45[0-5][0-7]验证 4500~4557 范围的八进制数值；4[0-4][0-7]{2}验证 4000~4477 范围的八进制数值；[1-3][0-7]{3}验证 1000~3777 范围的八进制数值；[1-7][0-7]{0,2}验证 1~777 范围的八进制数值。

使用工具 Regex Tester 测试正则表达式`^((456[0-7])|(45[0-5][0-7])|(4[0-4][0-7]{2})|([1-3][0-7]{0,3})|([1-7][0-7]{0,2}))$`，结果如图 2.21 所示。

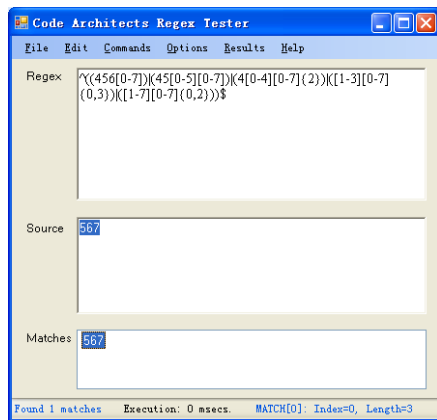


图 2.21 测试正则表达式`^((456[0-7])|(45[0-5][0-7])|(4[0-4][0-7]{2})|([1-3][0-7]{0,3})|([1-7][0-7]{0,2}))$`

2.1.9 十六进制数值的验证

十六进制数值可以包含 0、1、2、3、4、5、6、7、8、9、A、B、C、D、E 和 F 这 10 个数字和 6 个字母（也可以使用小写字母）。它的验证方法比较简单，以下正则表达式都能够验证十六进制数值。

```
^(0|([1-9A-Fa-f][0-9A-Fa-f]*))$ (103)
```

```
^(0|([1-9A-Fa-f][0-9A-Fa-f]{0,}))$ (104)
```

1. 验证指定范围的十六进制数值

验证给定范围的十六进制数值的方法比上述验证方法稍微复杂。在此，不妨设十六进制的范

围为 1~ABCD。为了验证该范围内的十六进制数值，特把该范围划分为以下 5 个范围。

- ❑ 1~FFF
- ❑ 1000~9FFF
- ❑ A000~AAFF
- ❑ AB00~ABBF
- ❑ ABC0~ABCD

以下正则表达式能够验证 1~FFF 范围的十六进制数值。

```
^[1-9A-Fa-f][0-9A-Fa-f]{0,2}$ (105)
```

以下正则表达式能够验证 1000~9FFF 范围的十六进制数值。

```
^[1-9][0-9A-Fa-f]{3}$ (106)
```

以下正则表达式能够验证 A000~AAFF 范围的十六进制数值。

```
^[aA][0-9Aa][0-9A-Fa-f]{2}$ (107)
```

以下正则表达式能够验证 AB00~ABBF 范围的十六进制数值。

```
^[aA][bB][0-9A-Ba-b][0-9A-Fa-f]$ (108)
```

以下正则表达式能够验证 ABC0~ABCD 范围的十六进制数值。

```
^[aA][bB][cC][0-9A-Da-d]$ (109)
```

综合以上，可以得到验证 1~ABCD 范围的十六进制数值的正则表达式，具体如下：

```
^((([aA][bB][cC][0-9A-Da-d])|([aA][bB][0-9A-Ba-b][0-9A-Fa-f])|([aA][0-9Aa][0-9A-Fa-f]{2})|([1-9][0-9A-Fa-f]{3})|([1-9A-Fa-f][0-9A-Fa-f]{0,2})))$ (110)
```

2. 示例（109）讲解

示例（109）的讲解如下。

- ❑ ^和\$分别匹配字符串的开始和结束位置。
- ❑ [aA][bB][cC][0-9A-Da-d]匹配 ABC0~ABCD 范围的十六进制数值，[0-9A-Da-d]可以匹配 0~D 范围的任何一位数字。
- ❑ [aA][bB][0-9A-Ba-b][0-9A-Fa-f]匹配 AB00~ABBF 范围的十六进制数值，[0-9A-Ba-b]可以匹配 0~B 范围的任何一位数字，[0-9A-Fa-f]匹配 0~F 范围的任何一位数字。
- ❑ [1-9][0-9A-Fa-f]{3}匹配 1000~9FFF 范围的十六进制数值，[0-9A-Fa-f]{3}可以匹配 3 位 0~FFF 范围的任何 3 位数字。
- ❑ [1-9A-Fa-f][0-9A-Fa-f]{0,2}可以匹配长度为 1、2 或者 3 的 1~FFF 范围内的任何一个十六进制数值。

使用工具 Regex Tester 测试正则表达式示例（110），结果如图 2.22 所示。

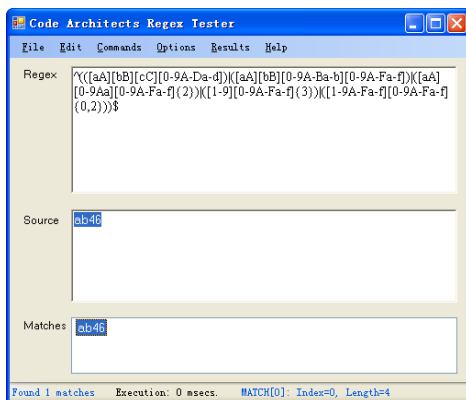


图 2.22 测试正则表达式示例（110）

2.2 4 种国内电话号码的验证

本节将介绍目前国内电话号码的验证方法，如手机号码、固定电话号码、区号+固定电话号码和区号+固定电话号码+分机号这4种电话号码的验证方法。

2.2.1 国内手机号码验证

目前国内的手机号码大多是以13或者15开头的、长度为11的数字字符串。其中，13开头的手机号码可以分为以130、131、132、133、134、135、136、137、138和139开头的手机号码。15开头的手机号码可以分为以158和159开头的手机号码。因此，在验证手机号码时也需要分两种情况分别验证。

以13开头的手机号码的组成规则为：13+9位数字字符串。以下正则表达式都能够验证以13开头的手机号码。

<code>\b13\d{9}\b</code>	(111)
<code>\b13[0-9]{9}\b</code>	(112)

以15开头的手机号码的组成规则为：15+8位数字字符串或者159+8位数字字符串。以下正则表达式都能够验证以15开头的手机号码。

<code>\b15[89]\d{8}\b</code>	(113)
<code>\b15[8-9]\d{8}\b</code>	(114)
<code>\b15[89][0-9]{8}\b</code>	(115)

综上所述，以下正则表达式能够验证目前国内大多数手机号码。

<code>\b(13\d{9}) (15[89]\d{8})\b</code>	(116)
--	-------

当用户所拨打的手机号码的所属地不是用户所在地时，需要在拨打手机号码前加拨一个0。因此，手机号码也可以是0开头、后接11位手机号码的数字字符串。以下正则表达式能够验证该类型的手机号码。

<code>\b0(13\d{9}) (15[89]\d{8})\b</code>	(117)
---	-------

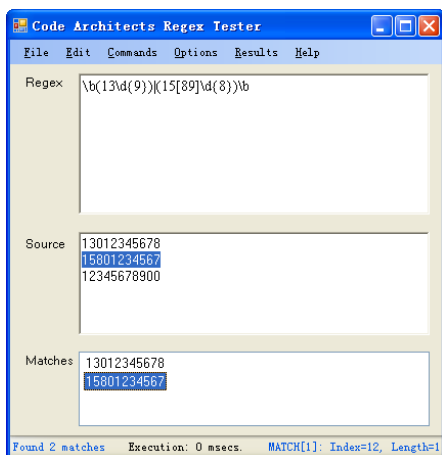
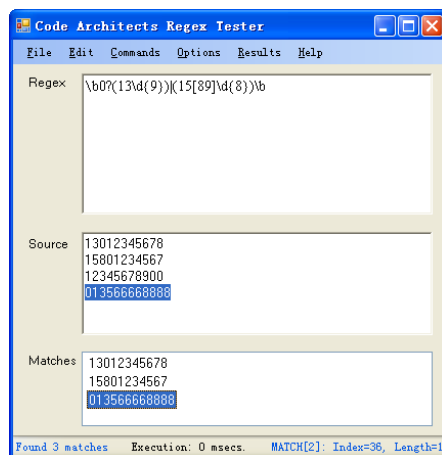
综合以上，以下正则表达式能够验证目前国内的（除18等开头的）所有手机号码（包括前面添加数字0的手机号码）。

<code>\b0?(13\d{9}) (15[89]\d{8})\b</code>	(118)
--	-------

正则表达式解释如下。

- 正则表达式**`\b(13\d{9})|(15[89]\d{8})\b`**。**`\b`**匹配单词的边界，即单词的开始位置或结束位置；**`(13\d{9})`**匹配以13开头的手机号码；**`(15[89]\d{8})`**匹配以15开头的手机号码。
- 正则表达式**`\b0?(13\d{9})|(15[89]\d{8})\b`**。**`\b`**匹配单词的边界，即单词的开始位置或结束位置；**`0?`**表示数字0可以出现一次或者不出现，当数字0出现时，则匹配前面加数字0的手机号码，否则匹配前面不加数字0的手机号码；**`(13\d{9})`**匹配以13开头的手机号码；**`(15[89]\d{8})`**匹配以15开头的手机号码。

使用工具 Regex Tester 分别测试正则表达式**`\b(13\d{9})|(15[89]\d{8})\b`**和**`\b0?(13\d{9})|(15[89]\d{8})\b`**，结果分别如图2.23和图2.24所示。

图 2.23 测试正则表达式 `\b(13\d{9})(15[89]\d{8})\b`图 2.24 测试正则表达式 `\b0?(13\d{9})(15[89]\d{8})\b`

2.2.2 固定电话号码（不包括区号）验证

目前国内的固定电话号码（不包括区号）一般为长度为 7 位或者 8 位的数字字符串，如 8888555、12344321 等。以下正则表达式都能够验证不包括区号的固定电话号码。

```
\b\d{7,8}\b (119)
```

```
\b[0-9]{7,8}\b (120)
```

然而，在某一个城市或者一个区域中，用户的固定电话号码往往在一个区段里面，如 23730001~23759999 等。此时，如果仍然使用正则表达式（119）验证该区段内的固定号码，则是相当不准确的。下面就以验证 23730001~23759999 区段电话号码为例，来介绍指定区段的固定号码验证。为了验证该区段的电话号码，把 23730001~23759999 区段的电话号码分为以下 5 个区段。

- ❑ 23730001~23730009
- ❑ 23730010~23730099
- ❑ 23730100~23730999
- ❑ 23731000~23739999
- ❑ 23740000~23759999

以下正则表达式都能够验证 23730001~23730009 区段的电话号码。

```
\b2373000[1-9]\b (121)
```

```
\b2373000[123456789]\b (122)
```

```
\b2373000(1|2|3|4|5|6|7|8|9)\b (123)
```

以下正则表达式能够验证 23730010~23730099 区段的电话号码。

```
\b237300[1-9]\d\b (124)
```

以下正则表达式能够验证 23730100~23730999 区段的电话号码。

```
\b23730[1-9]\d{2}\b (125)
```

以下正则表达式能够验证 23731000~23739999 区段的电话号码。

```
\b2373[1-9]\d{3}\b (126)
```

以下正则表达式能够验证 23740000~23759999 区段的电话号码。

```
\b237[4-5]\d{4}\b (127)
```

```
\b237[45]\d{4}\b (128)
```

综上所述，以下正则表达式都能够验证 23730001~23759999 区段的固定电话号码。

```
\b237(((45)\d{4})|(3[1-9]\d{3})|(30[1-9]\d{2})|(300[1-9]\d)|(3000[1-9]))\b
(129)
\b237(((45)\d{4})|3(((1-9)\d{3})|(0[1-9]\d{2})|(00[1-9]\d)|(000[1-9])))\b
(130)
```

1. 示例（129）讲解

示例（129）的讲解如下。

- ❑ \b 匹配单词的边界。
- ❑ 237 匹配电话号码的开头 3 位。
- ❑ ([45]\d{4}) 匹配 23740000～23759999 区段内的电话号码的后 5 位。
- ❑ 3[1-9]\d{3} 匹配 23731000～23739999 区段内的电话号码的后 5 位。
- ❑ 30[1-9]\d{2} 匹配 23730100～23730999 区段内的电话号码的后 5 位。
- ❑ 300[1-9]\d 匹配 23730010～23730099 区段内的电话号码的后 5 位。
- ❑ (3000[1-9]) 匹配 23730001～23730009 区段内的电话号码的后 5 位。

2. 示例（130）讲解

示例（130）的讲解如下。

- ❑ \b 匹配单词的边界。
- ❑ 237 匹配电话号码的开头 3 位。
- ❑ ([45]\d{4}) 匹配 23740000～23759999 区段内的电话号码的后 5 位。
- ❑ 3(((1-9)\d{3})|(0[1-9]\d{2})|(00[1-9]\d)|(000[1-9])) 中第一个 3 匹配 2373 中的第二个数字 3。
- ❑ [1-9]\d{3} 匹配 23731000～23739999 区段内的电话号码的后 4 位。
- ❑ 0[1-9]\d{2} 匹配 23730100～23730999 区段内的电话号码的后 4 位。
- ❑ 00[1-9]\d 匹配 23730010～23730099 区段内的电话号码的后 4 位。
- ❑ (000[1-9]) 匹配 23730001～23730009 区段内的电话号码的后 4 位。

使用工具 Regex Tester 分别测试正则表达式示例（129）和示例（130），结果分别如图 2.25 和图 2.26 所示。

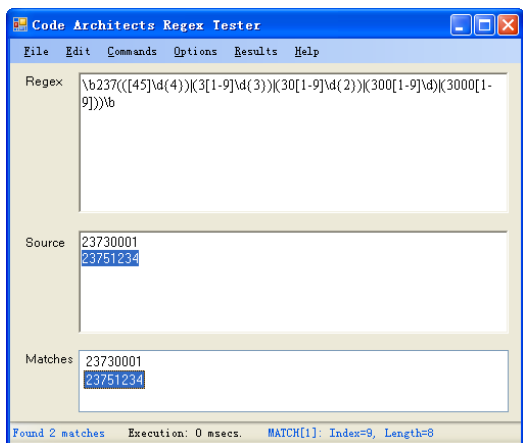


图 2.25 测试正则表达式示例（129）

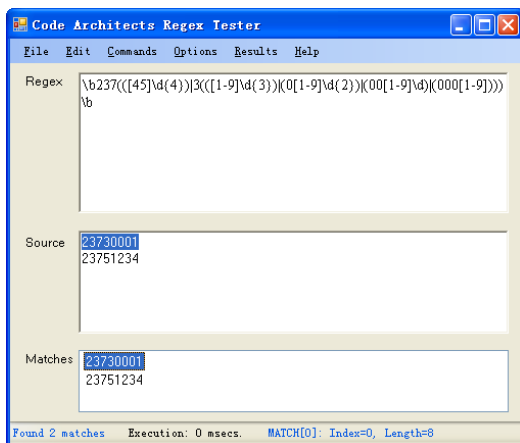


图 2.26 测试正则表达式示例（130）

2.2.3 区号+固定电话号码验证

目前，国内的电话号码包括三部分：区号、固定电话号码和连接符（如连接字符-、空白字符

等)。其中，区号是长度为 3 位或者 4 位的数字字符串；固定电话号码为长度为 7 位或者 8 位的数字字符串。要想验证目前国内的电话号码，可以分别验证区号部分和固定电话号码部分。

区号部分一般是以 0 开头、长度为 3 位或 4 位的数字字符串。以下正则表达式都能够验证区号部分。

```
\b0\d{2,3}\b (131)
```

```
\b0[0-9]{2,3}\b (132)
```

固定电话号码部分一般为长度等于 7 或者 8 的数字字符串。以下正则表达式都能够验证固定电话号码部分。

```
\b\d{7,8}\b (133)
```

```
\b[0-9]{7,8}\b (134)
```

综合以上，以下正则表达式都能够验证区号+固定电话号码。

```
\b0\d{2,3}[- ]?\d{7,8}\b (135)
```

```
\b0[0-9]{2,3}[- ]?[0-9]{7,8}\b (136)
```

1. 示例（135）讲解

示例（135）的讲解如下。

- \b 匹配单词的边界。
- 0 匹配区号部分的第一位数字 0。
- \d{2,3} 匹配区号部分的后面两位或者三位的数字字符串。
- [-]? 可以匹配一个空白符号或者连接符号，或者匹配不存在空白符号和连接符号的字符串。
- \d{7,8} 匹配固定电话号码部分的 7 位或者 8 位的数字字符串。

2. 示例（136）讲解

示例（136）的讲解如下。

- \b 匹配单词的边界。
- 0 匹配区号部分的第一位数字 0。
- [0-9]{2,3} 匹配区号部分的后面两位或者三位的数字字符串。
- [-]? 可以匹配一个空白符号或者连接符号，或者匹配不存在空白符号和连接符号的字符串。
- [0-9]{7,8} 匹配固定电话号码部分的 7 位或者 8 位的数字字符串。

使用工具 Regex Tester 测试正则表达式 `\b0\d{2,3}[-]?\d{7,8}\b`，结果如图 2.27 所示。



图 2.27 测试正则表达式 `\b0\d{2,3}[-]?\d{7,8}\b`

2.2.4 区号+固定电话号码+分机号码验证

一些比较大的公司、企业或政府部门在向外部提供固定电话号码时，除了区号、固定电话号码之外，可能还包括分机号码。下面就来介绍这种格式的电话号码的验证方法。

分机号码的编码规则一般由各公司、企业或政府部门的内部规定，不同公司、企业或政府部门的分机号码可能是不相同的。在此，不妨考虑长度为4位的分机号码。设区号、固定电话号码的字符串为X，分机号码的字符串为Y，则整个电话号码的可能构成如下。

- X-Y
- XY
- XY

其中，X部分的验证方法已经介绍，在此不再介绍。Y部分的验证比较简单。以下正则表达式都能够验证分机号码部分。

<code>[-]?\d{4}\b</code>	(137)
<code>[-]?[0-9]{4}\b</code>	(138)

综合以上，以下正则表达式都能够验证区号+固定电话号码+分机号码。

<code>\b0\d{2,3}[-]?\d{7,8}[-]?\d{4}\b</code>	(139)
<code>\b0[0-9]{2,3}[-]?[0-9]{7,8}[-]?[0-9]{4}\b</code>	(140)

1. 示例(139)讲解

示例(139)的讲解如下。

- `\b` 匹配单词的边界。
- `0` 匹配区号部分的第一位数字0。
- `\d{2,3}` 匹配区号部分的后面两位或者三位的数字字符串。
- `[-]?` 可以匹配一个空白符号或者连接符号，或者匹配不存在空白符号和连接符号的字符串，该表达式还用来匹配固定电话号码和分机号码的连接符号。
- `\d{7,8}` 匹配固定电话号码部分的7位或者8位的数字字符串。
- `d{4}` 匹配4位分机号码。

2. 示例(140)讲解

示例(140)的讲解如下。

- `\b` 匹配单词的边界。
- `0` 匹配区号部分的第一位数字0。
- `[0-9]{2,3}` 匹配区号部分的后面两位或者三位的数字字符串。
- `[-]?` 可以匹配一个空白符号或者连接符号，或者匹配不存在空白符号和连接符号的字符串，该表达式还用来匹配固定电话号码和分机号码的连接符号。
- `[0-9]{7,8}` 匹配固定电话号码部分的7位或者8位的数字字符串。
- `d{4}` 匹配4位分机号码。

注意：正则表达式 `\b0\d{2,3}[-]?\d{7,8}[-]?\d{4}\b` 能够匹配形如 010-12345678 4563 的电话号码。该电话号码的区号、固定电话号码之间的连接符号和固定电话号码、分机号码之间的连接符号可以相同，也可以不同。如果要求电话号码中的这两个连接符号相同，则需要使用后向引用来验证满足该要求的电话号码。

以下正则表达式都能够验证区号+固定电话号码+分机号码，且区号、固定电话号码之间的

连接符号和固定电话号码、分机号码之间的连接符号相同。

```
\b0\d{2,3}(?<char>[- ]?)\d{7,8}\k<char>\d{4}\b (141)
```

```
\b0[0-9]{2,3}(?<char>[- ]?)[0-9]{7,8}\k<char>[0-9]{4}\b (142)
```

3. 示例（141）讲解

示例（141）的讲解如下。

- ❑ \b 匹配单词的边界。
- ❑ 0 匹配区号部分的第一位数字 0。
- ❑ \d{2,3} 匹配区号部分的后面两位或者三位的数字字符串。
- ❑ (?<char>[-]?) 是一个名称为 “char” 的分组。该分组可以匹配一个空白符号或者连接符号，或者匹配不存在空白符号和连接符号的字符串，并将匹配的内容命名为 “char”。
- 该表达式还用来匹配固定电话号码和分机号码的连接符号。
- ❑ \d{7,8} 匹配固定电话号码部分的 7 位或者 8 位的数字字符串。
- ❑ \k<char> 后向引用名称为 “char” 的分组匹配的内容。
- ❑ d{4} 匹配 4 位分机号码。

使用工具 Regex Tester 分别测试正则表达式示例（141）和示例（142），结果分别如图 2.28 和图 2.29 所示。

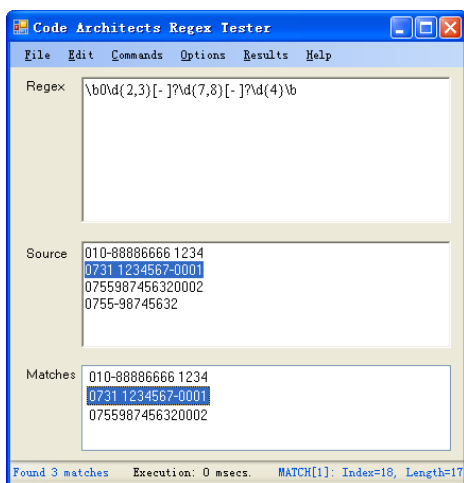


图 2.28 测试正则表达式示例（141）

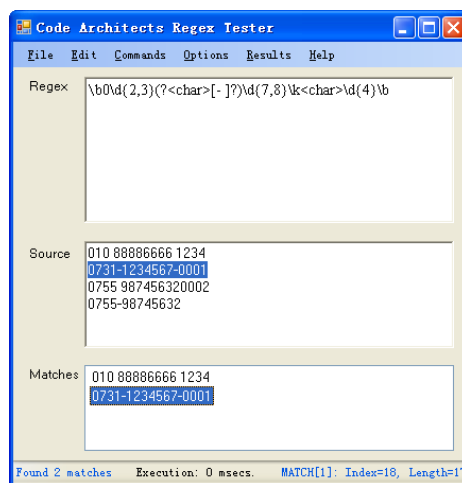


图 2.29 测试正则表达式示例（142）

2.2.5 固定电话号码验证总结

2.2.2 节、2.2.3 节和 2.2.4 节分别介绍了三种不同情况的固定电话号码的验证方法。若被验证的电话号码既可以不包括区号，也可以不包括分机号码，则把这种类型的电话号码称为通用固定电话号码。如果需要验证通用固定电话号码，则只需要综合 2.2.2 节、2.2.3 节和 2.2.4 节中的各种类型的验证即可。以下正则表达式能够验证通用固定电话号码。

```
\b(0\d{2,3}(?<char>[- ]?)\d{7,8}\k<char>\d{4})|((0\d{2,3}[- ]?\d{7,8})|(\d{7,8}))\b (143)
```

然而，上述正则表达式过于复杂。以下正则表达式虽然比上述正则表达式要简单许多，但它也能够验证通用固定电话号码。

```
\b(0\d{2,3}(?<char>[- ]?))?\d{7,8}(\k<char>\d{4})?\b (144)
```


1. 示例（143）讲解

示例（143）的讲解如下。

- `(0\d{2,3})(?<char>[-]?)\d{7,8}\k<char>\d{4})`验证区号+固定电话号码+分机号码类型的电话号码。
- `(0\d{2,3}[-]?\d{7,8})`验证区号+固定电话号码类型的电话号码。
- `(\d{7,8})`验证固定电话号码类型的电话号码。

注意：正则表达式`\b(0\d{2,3})(?<char>[-]?)\d{7,8}\k<char>\d{4})|(0\d{2,3}[-]?\d{7,8})|(\d{7,8})\b`是简单地组合了验证三种不同情况电话号码的正则表达式。

2. 示例（144）讲解

示例（144）的讲解如下。

- `(0\d{2,3})(?<char>[-]?)?`用来验证存在区号和不存在区号的两种情况的电话号码。
- `0\d{2,3}`验证长度为3或者4的区号。
- `(?<char>[-]?)`验证区号和固定电话号码之间的连接符号。
- `(\k<char>\d{4})?`验证存在分机号码和不存在分机号码的两种情况的电话号码。如果存在分机号码，则验证分机号码和固定号码之间的连接符号、区号和固定电话号码之间的连接符号是否相同。
- `\d{7,8}`验证长度为7或者8的固定电话号码。
- `\d{4}`验证长度为4的分机号码。

注意：正则表达式`\b(0\d{2,3})(?<char>[-]?)?\d{7,8}(\k<char>\d{4})?\b`是从区号或者分机号码是否存在这个角度来考虑通用固定电话号码的验证方法。

使用工具 Regex Tester 分别测试正则表达式示例（143）和示例（144），结果分别如图 2.30 和图 2.31 所示。

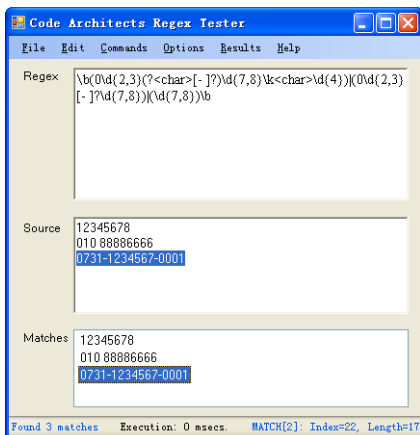


图 2.30 测试正则表达式示例（143）

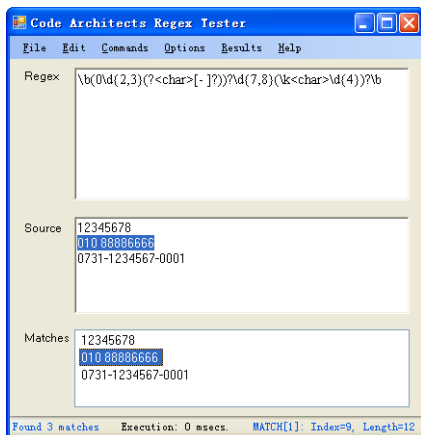


图 2.31 测试正则表达式示例（144）

2.3 2 种身份证号码验证

目前国内公民身份证号码一般为 15 位或 18 位，从左至右依次为：6 位地址码、6 位或 8 位

出生日期码、3 位顺序码和 1 位校验码（只在 18 位号码中存在）。其具体含义如下。

- ❑ 地址码：表示编码对象常住户口所在县（市、旗、区）的行政区划代码，按 GB/T2260 的规定执行。
- ❑ 出生日期码：表示编码对象出生的年、月、日，按 GB/T7408 的规定执行。年、月、日分别用 4 位（15 身份证号码使用 2 位表示年）、2 位、2 位数字表示，之间不用分隔符。
- ❑ 顺序码：表示在同一地址码所标识的区域范围内，对同年、同月、同日出生的人编定的顺序号，顺序码的奇数分配给男性，偶数分配给女性。

其中，15 位身份证号码的出生日期码使用 2 位表示年，18 位身份证号码的出生日期码使用 4 位表示年。校验码可以为 0、1、2、3、4、5、6、7、8、9、X。其中最后一个为字母 X 表示 10。下面分别介绍 15 位身份证号码和 18 位身份证号码的验证方法。

2.3.1 15 位身份证号码验证

15 位身份证号码从左至右依次为：6 位地址码、6 位出生日期码和 3 位顺序码。其中，6 位地址码可以是长度为 6 的任意数字字符串；6 位出生日期码由 2 位年编码、2 位月编码和 2 位日编码组成；3 位顺序码可以是长度为 3 的任意数字字符串。

在 6 位出生日期码中，2 位月编码应该在 01~12 范围之内，2 位日编码应该在 01~31 范围之内。因此，以下正则表达式能够验证 6 位出生日期码。

```
\d{2}((0\d)|(1[0-2]))((3[01])|([0-2]\d)) (145)
```

综上所述，以下正则表达式能够验证 15 位身份证号码。

```
^\d{8}((0\d)|(1[0-2]))((3[01])|([0-2]\d))\d{3}$ (146)
```

正则表达式解释如下。

- ❑ 正则表达式 `(0\d)|(1[0-2])` 验证身份证号码中的两位月编码，能够被该表达式验证的字符串应该在 01~12 范围之内。
- ❑ 正则表达式 `(3[01])|([0-2]\d)` 验证身份证号码中的两位日编码，能够被该表达式验证的字符串应该在 01~31 范围之内。

使用工具 Regex Tester 测试正则表达式 `^\d{8}((0\d)|(1[0-2]))((3[01])|([0-2]\d))\d{3}$`，结果如图 2.32 所示。

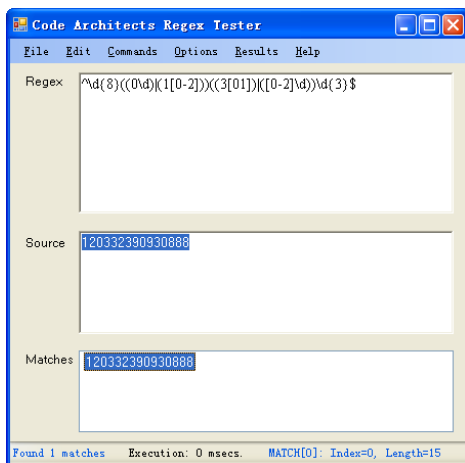


图 2.32 测试正则表达式 `^\d{8}((0\d)|(1[0-2]))((3[01])|([0-2]\d))\d{3}$`

2.3.2 18 位身份证号码验证

18 位身份证号码从左至右依次为：6 位地址码、8 位出生日期码、3 位顺序码和 1 位校验码。其中，6 位地址码可以是长度为 6 的任意数字字符串；8 位出生日期码由 4 位年编码、2 位月编码、2 位日编码组成；3 位顺序码的前 3 位可以是长度为 3 的任意数字字符串；1 位校验码可以是任意数字或大写字母 X。

在 8 位出生日期码中，4 位年编码（目前来说）应该为 18XX、19XX 和 2XXX 的形式，2 位月编码应该在 01~12 范围之内，2 位日编码应该在 01~31 范围之内。因此，以下正则表达式能够验证 8 位出生日期码。

```
((1[89])|(2\d))\d{2}((0\d)|(1[0-2]))((3[01])|([0-2]\d)) (147)
```

综上所述，以下正则表达式能够验证 18 位身份证号码。

```
^\d{6}((1[89])|(2\d))\d{2}((0\d)|(1[0-2]))((3[01])|([0-2]\d))\d{3}(\d|X)$ (148)
```

正则表达式解释如下。

- 正则表达式`((1[89])|(2\d))\d{2}`验证 4 位年编码，其中，年编码的格式为 18XX、19XX 和 2XXX。
- 正则表达式`(0\d)|(1[0-2])`验证身份证号码中的 2 位月编码，能够被该表达式验证的字符串应该在 01~12 范围之内。
- 正则表达式`(3[01])|([0-2]\d)`验证身份证号码中的 2 位日编码，能够被该表达式验证的字符串应该在 01~31 范围之内。
- 正则表达式`\d{3}(\d|X)`验证 3 位顺序码和 1 位校验码，其中校验码可以是数字或大写字母 X。

使用工具 Regex Tester 测试正则表达式`^\d{6}((1[89])|(2\d))\d{2}((0\d)|(1[0-2]))((3[01])|([0-2]\d))\d{3}(\d|X)$`，结果如图 2.33 所示。

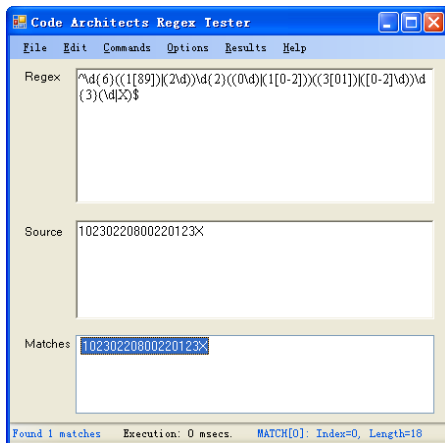


图 2.33 测试正则表达式`^\d{6}((1[89])|(2\d))\d{2}((0\d)|(1[0-2]))((3[01])|([0-2]\d))\d{3}(\d|X)$`

2.4 银行卡和信用卡号验证

本节将介绍目前普通银行卡卡号和信用卡卡号的编码规则，以及它们的验证方法。普通银行卡卡号通常为 13~19 位，而信用卡卡号一般为 16 位。

1. 信用卡卡号验证

目前多数银行的信用卡卡号一般由 16 位数字组成，并遵循国际标准。信用卡卡号的前 6 位由 BIN 分配，其中，第一位为 4 的表示为 VISA 卡、为 5 的是 Master 卡；卡号的第 7 至 15 位由各个发卡行自己定义；最后一位，即第 16 位，为校验位。以下正则表达式能够验证长度为 16 位的信用卡卡号。

```
^[4-5]\d{15}$ (149)
```

```
^[4-5][0-9]{15}$ (150)
```

2. 建行卡号验证

目前，建设银行卡号一般由 19 位数字组成，并遵循国际标准。最常见的卡号是以“436742”或者“622700”开头，其余 13 位由银行自行制定。以下正则表达式能够验证长度为 19 位的建设银行卡号。

```
^((436742)|(622700))\d{13}$ (151)
```

```
^((436742)|(622700))[0-9]{13}$ (152)
```

3. 工行卡号验证

目前，工商银行卡号一般由 19 位数字组成，并遵循国际标准。最常见的卡号是以“955880”或者“402791”开头，其余 13 位由银行自行制定。以下正则表达式能够验证长度为 19 位的工商银行卡号。

```
^((955880)|(402791))\d{13}$ (153)
```

```
^((955880)|(402791))[0-9]{13}$ (154)
```

4. 农行卡号验证

目前，农业银行卡号一般由 19 位数字组成，并遵循国际标准。最常见的卡号是以“955998”开头，其余 13 位由银行自己制定。以下正则表达式能够验证长度为 19 位的农业银行卡号。

```
^955998\d{13}$ (155)
```

```
^955998[0-9]{13}$ (156)
```

2.5 邮政编码验证

本节介绍邮政编码的验证方法，如国内邮政编码、美国邮政编码和日本邮政编码等邮政编码的验证方法。

2.5.1 国内邮政编码验证

目前国内邮政编码采用四级六位编码制：前两位表示省、市、自治区，第三位代表邮区，第四位代表县、市，最后两位代表投递邮局，即代表从这个城市哪个投递区投递的，是投递区的位置。例如，邮政编码“130021”，“13”代表吉林省，“00”代表省会长春，“21”代表所在投递区。

国内邮政编码是一个长度为 6 的数字字符串。以下正则表达式都能够验证国内邮政编码。

```
^\d{6}$ (157)
```

```
^[0-9]{6}$ (158)
```

正则表达式解释如下。

- 正则表达式`^\d{6}$`：`^`和`$`分别验证字符串的开始和结束位置；`\d{6}`验证邮政编码的 6 位数字。

- 正则表达式`^[0-9]{6}$`：`^`和`$`分别验证字符串的开始和结束位置；`[0-9]{6}`验证邮政编码的6位数字。

2.5.2 国际邮政编码验证

在国际上，每个国家都有自己的邮政编码的编码规则。邮政编码往往由字母或数字构成。下面就来验证美国邮政编码和日本邮政编码。

1. 美国邮政编码

目前美国邮政编码的编码可以仅由5个数字组成，或者由5位数字、连接符`-`、4位数字组成。第一位数字为州的代码，第2位和第3位数字表示地区，第4位和第5位数字表示具体的区域，最后4位表示邮件被投递的具体地址（如果存在）。以下正则表达式都能够验证美国邮政编码。

<code>^\d{5}(-\d{4})?\$</code>	(159)
<code>^(\d{5}-\d{4}) \d{5}\$</code>	(160)

正则表达式解释如下。

- 正则表达式`^\d{5}(-\d{4})?$`：`^`和`$`分别验证字符串的开始和结束位置；`\d{6}`验证5位数字字符串；`(-\d{4})?`可以带连接字符`-`的4位数字字符串，该字符串可以出现1次或者不出现，若出现1次，则验证9位邮政编码，否则为5位邮政编码。
- 正则表达式`^(\d{5}-\d{4})|\d{5}$`：`^`和`$`分别验证字符串的开始和结束位置；`\d{5}-\d{4}`匹配9位的邮政编码`[0-9]{6}`；`\d{5}`匹配5位的邮政编码。

2. 日本邮政编码

目前日本邮政编码的编码由7个数字组成，其中，前3位和后4位使用连接符`-`进行连接。以下正则表达式都能够验证日本邮政编码。

<code>^\d{3}-\d{4}\$</code>	(161)
<code>^[0-9]{3}-[0-9]{4}\$</code>	(162)

正则表达式解释如下。

- 正则表达式`^\d{3}-\d{4}$`：`^`和`$`分别验证字符串的开始和结束位置；`\d{3}`匹配3位数字字符串；`-`匹配连接符号；`\d{4}`匹配4位数字字符串。
- 正则表达式`^[0-9]{3}-[0-9]{4}$`：`^`和`$`分别验证字符串的开始和结束位置；`[0-9]{3}`匹配3位数字字符串；`-`匹配连接符号；`[0-9]{4}`匹配4位数字字符串。

2.6 4种IP地址验证

本节介绍IP地址的验证方法，包括简单IP地址、精确IP地址、子网内部IP地址和64位IP地址的验证方法（前面三种类型的IP地址都是32位IP地址）。

2.6.1 简单IP地址验证

简单IP地址组成规则为：1~3位整数.1~3位整数.1~3位整数.1~3位整数。因此，要验证简单IP地址，首先需要验证1~3位整数。以下正则表达式都能够验证1~3位整数。

<code>[1-9]\d{0,2}</code>	(163)
<code>[1-9][0-9]{0,2}</code>	(164)

综上所述，以下正则表达式都能够验证简单IP地址。

```
^([1-9]\d{0,2}\.){3}[1-9]\d{0,2}$ (165)
^([1-9][0-9]{0,2}\.){3}[1-9][0-9]{0,2}$ (166)
```

1. 示例（167）讲解

示例（165）的讲解如下。

- ^和\$分别验证字符串的开始和结束位置。
- ([1-9]\d{0,2}\.){3}中的[1-9]\d{0,2}验证 1~3 位整数，\验证字符。
- ([1-9]\d{0,2}\.){3}表示 3 位整数和字符.可以出现 3 次。

2. 示例（166）讲解

示例（166）的讲解如下。

- ^和\$分别验证字符串的开始和结束位置。
- ([1-9][0-9]{0,2}\.){3}中的[1-9][0-9]{0,2}验证 1~3 位整数，\验证字符。
- ([1-9][0-9]{0,2}\.){3}表示 3 位整数和字符.可以出现 3 次。

使用工具 Regex Tester 测试正则表达式^([1-9]\d{0,2}\.){3}[1-9]\d{0,2}\$，结果如图 2.34 所示。

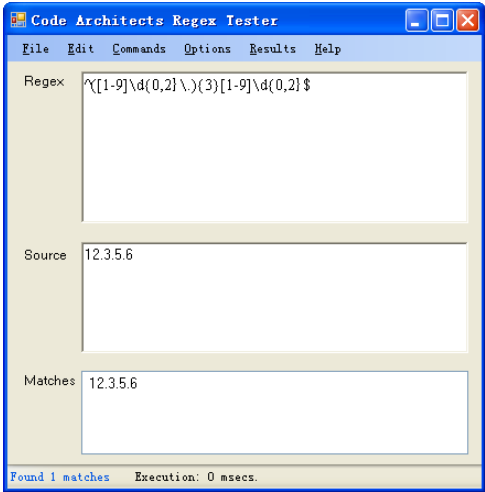


图 2.34 测试正则表达式^([1-9]\d{0,2}\.){3}[1-9]\d{0,2}\$

2.6.2 精确 IP 地址验证

正则表达式^([1-9]\d{0,2}\.){3}[1-9]\d{0,2}\$可以验证简单 IP 地址。同时，它也能够匹配形如 666.777.888.999 的 IP 地址。显然，这个 IP 地址是不合法的。在 32 位 IP 地址中，每一个数值都应该是在 0~255 范围内。因此，在 IP 地址的编码规则（1~3 位整数.1~3 位整数.1~3 位整数.1~3 位整数）中，每一个整数都应该在 0~255 范围内。在此，把该类型的 IP 地址的验证称为精确 IP 地址验证。

要验证精确 IP 地址，首先需要验证 0~255 范围内的整数。在此，把 0~255 范围划分为 4 个范围：0~99、100~199、200~249、250~255。

以下正则表达式都能够验证 0~99 范围内的整数。

```
([1-9]\d?)|0 (167)
([1-9][0-9]?)|0 (168)
```

以下正则表达式都能够验证 100~199 范围内的整数。

<code>1\d{2}</code>	(169)
<code>1[0-9]{2}</code>	(170)

以下正则表达式都能够验证 200~249 范围内的整数。

<code>2[0-4]\d</code>	(171)
<code>2[0-4][0-9]</code>	(172)

以下正则表达式能够验证 250~255 范围内的整数。

<code>25[0-5]</code>	(173)
----------------------	-------

综上所述, 以下正则表达式能够验证 0~255 范围内的整数。

<code>(25[0-5]) (2[0-4]\d) (1\d{2}) ([0-9])</code>
--

如果把 0~99 和 100~199 合并为一个范围, 即 0~199, 则以下正则表达式能够验证 0~199 范围内的整数。

<code>[01]?\d\d?</code>	(174)
-------------------------	-------

因此, 以下正则表达式也能够验证 0~255 范围内的整数。

<code>(25[0-5]) (2[0-4]\d) ([01]?\d\d?)</code>
--

综上所述, 以下正则表达式都能够验证精确 IP 地址。

<code>^((25[0-5]) (2[0-4]\d) (1\d{2}) ([0-9])\.)\{3\}((25[0-5]) (2[0-4]\d) (1\d{2}) ([0-9]))\$</code>	(175)
---	-------

<code>^((25[0-5]) (2[0-4]\d) ([01]?\d\d?))\.)\{3\}((25[0-5]) (2[0-4]\d) ([01]?\d\d?))\$</code>	(176)
--	-------

1. 示例 (175) 讲解

示例 (175) 的讲解如下。

- ^和\$分别验证字符串的开始和结束位置。
- `(25[0-5])|(2[0-4]\d)|(1\d{2})|([0-9])`验证 0~255 范围内的整数, 其中, `25[0-5]`验证 250~255 范围内的整数、`2[0-4]\d`验证 200~249 范围内的整数、`1\d{2}`验证 100~199 范围内的整数、`[0-9]`验证 0~99 范围内的整数。
- \.验证字符。
- `((25[0-5])|(2[0-4]\d)|(1\d{2})|([0-9]))\.)\{3\}`表示 0~255 范围内的整数和字符.可以出现 3 次。

2. 示例 (176) 讲解

示例 (176) 的讲解如下。

- ^和\$分别验证字符串的开始和结束位置。
- `(25[0-5])|(2[0-4]\d)|([01]?\d\d?)`验证 0~255 范围内的整数, 其中, `25[0-5]`验证 250~255 范围内的整数、`2[0-4]\d`验证 200~249 范围内的整数、`[01]?\d\d?`验证 0~199 范围内的整数。
- \.验证字符。
- `((25[0-5])|(2[0-4]\d)|([01]?\d\d?))\.)\{3\}`表示 0~255 范围内的整数和字符.可以出现 3 次。

使用工具 Regex Tester 分别测试正则表达式示例 (175) 和示例 (176), 结果分别如图 2.35 和图 2.36 所示。

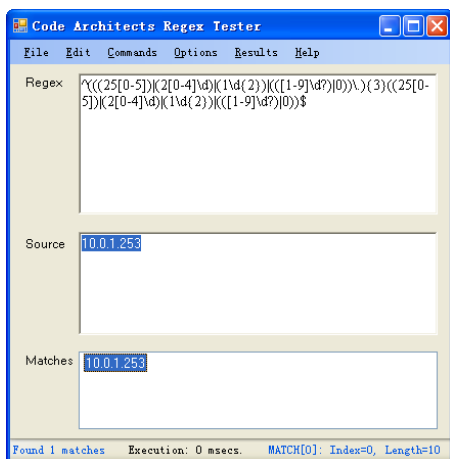


图 2.35 测试正则表达式示例 (175)

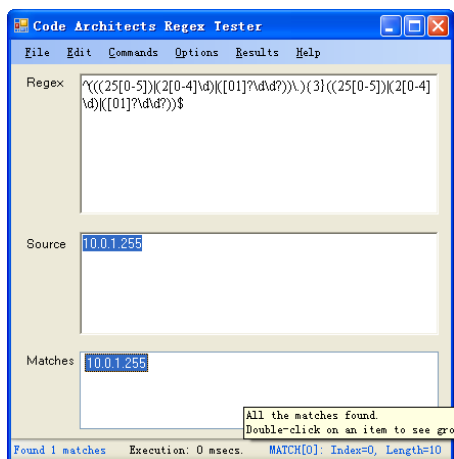


图 2.36 测试正则表达式示例 (176)

2.6.3 子网内部 IP 地址验证

根据不同的子网掩码可以把网络划分为多个不同的子网。为了方便下面介绍的验证方法，这里设子网掩码为 255.255.0.0（即 11111111111111110000000000000000），对外的 IP 地址为 192.168.0.0。因此，该子网内的 IP 地址范围为 192.168.0.0~192.168.255.255。

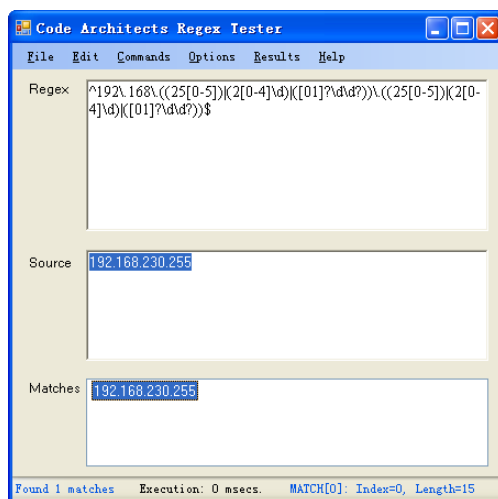
前面小节中已经介绍了精确 IP 地址的验证方法，以下正则表达式能够验证 0~255 范围内的整数。

```
(25[0-5])|(2[0-4]\d)|([01]?[0-9])$ (177)
```

综上所述，以下正则表达式能够验证 192.168.0.0~192.168.255.255 范围内的 IP 地址。

```
^192\.168\.((25[0-5])|(2[0-4]\d)|([01]?[0-9]))\.((25[0-5])|(2[0-4]\d)|([01]?[0-9]))$ (178)
```

使用工具 Regex Tester 测试正则表达式`^192\.168\.((25[0-5])|(2[0-4]\d)|([01]?[0-9]))\.((25[0-5])|(2[0-4]\d)|([01]?[0-9]))$`，结果如图 2.37 所示。

图 2.37 测试正则表达式`^192\.168\.((25[0-5])|(2[0-4]\d)|([01]?[0-9]))\.((25[0-5])|(2[0-4]\d)|([01]?[0-9]))$`

2.6.4 64 位 IP 地址验证

64 位 IP 地址一般被标记为 8 组 4 位的 16 进制数，且各组之间使用冒号 (:) 分割。下面就是一个 64 位 IP 地址的表示方法：

```
8000:0000:0000:0000:0123:4567:89AB:CDEF
```

因为很多 64 位 IP 地址中都有不少数字 0，在书写时较为不便。因此，64 位 IP 地址存在下面 3 种优化书写方式。

- 一组中的前几个 0 可以取消，如 0123 可以写成 123。
- 一组或多组全 0，可以用一对冒号 (::) 代替。因此，上面的 64 位 IP 地址可以写成 8000::123:4567:89AB:CDEF。
- 目前的 32 位 IP 地址可以写成一对冒号和一个老的带点的十进制数，如::192.168.2.052。

1. 验证形如 8000:0000:0000:0000:0123:4567:89AB:CDEF 的 IP 地址

在形如 8000:0000:0000:0000:0123:4567:89AB:CDEF 的 IP 地址中，每一组字符串的长度为 4，且每一组字符串都在 0000~FFFF 范围之内。以下正则表达式能够验证在 0000~FFFF 范围的十六进制数值。

```
[0-9A-F]{4} (179)
```

综上所述，以下正则表达式能够验证形如 8000:0000:0000:0000:0123:4567:89AB:CDEF 的 IP 地址。

```
^([0-9A-F]{4}:){7}[0-9A-F]{4}$ (180)
```

正则表达式解释如下。

- 正则表达式`^([0-9A-F]{4}:){7}[0-9A-F]{4}$`：`^`和`$`分别验证字符串的开始和结束位置；`[0-9A-F]{4}`验证在 0000~FFFF 范围的十六进制数值；`([0-9A-F]{4}:){7}`可以验证十六进制数值和冒号的字符串重复出现 7 次。

使用工具 Regex Tester 测试正则表达式`^([0-9A-F]{4}:){7}[0-9A-F]{4}$`，结果如图 2.38 所示。

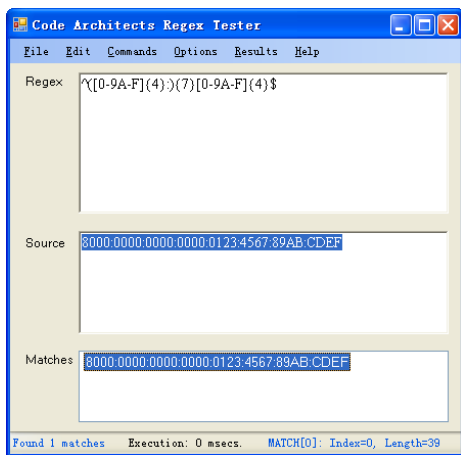


图 2.38 测试正则表达式`^([0-9A-F]{4}:){7}[0-9A-F]{4}$`

2. 验证形如 8000::123:4567:89AB:CDEF 的 IP 地址

在形如 8000::123:4567:89AB:CDEF 的 IP 地址中，每一组字符串的长度为 1~4，且每一组字符串都在 1~FFFF 范围之内。以下正则表达式能够验证 1~FFFF 范围内的十六进制数值。

```
[1-9A-F][0-9A-F]{0,3} (181)
```

如果 64 位 IP 地址中不存在双冒号，或者存在的双冒号在中间，则以下正则表达式能够验证该类型的 IP 地址。

```
^([1-9A-F][0-9A-F]{0,3}(:|:))){0,6}[1-9A-F][0-9A-F]{0,3}$ (182)
```

如果 64 位 IP 地址中的开头处存在双冒号，则以下正则表达式能够验证该类型的 IP 地址。

```
^::([1-9A-F][0-9A-F]{0,3}(:|:))){0,5}[1-9A-F][0-9A-F]{0,3}$ (183)
```

综上所述，以下正则表达式能够验证形如 8000::123:4567:89AB:CDEF 的 IP 地址。

```
^((([1-9A-F][0-9A-F]{0,3}(:|:))){0,6})|(::([1-9A-F][0-9A-F]{0,3}(:|:))){0,5})[1-9A-F][0-9A-F]{0,3}$ (184)
```

3. 示例（184）讲解

示例（184）的讲解如下。

- ❑ 正则表达式`[1-9A-F][0-9A-F]{0,3}`验证 1~FFFF 范围的十六进制数值。
- ❑ 正则表达式`([1-9A-F][0-9A-F]{0,3}(:|:)){0,6}`表示 1~FFFF 范围的十六进制数值和后面有冒号或者双冒号的字符串可以不出现，或者重复 1~6 次。
- ❑ 正则表达式`([1-9A-F][0-9A-F]{0,3}(:|:)){0,6}[1-9A-F][0-9A-F]{0,3}`验证形如 8000::123:4567:89AB:CDEF 的 IP 地址，且双冒号不能出现在 IP 地址的开头。
- ❑ 在正则表达式`^:([1-9A-F][0-9A-F]{0,3}(:|:)){0,5}[1-9A-F][0-9A-F]{0,3}$`中，双冒号出现在字符串的开头位置，`([1-9A-F][0-9A-F]{0,3}(:|:)){0,5}`表示 1~FFFF 范围的十六进制数值和后面有冒号或者双冒号的字符串可以不出现，或者重复 1~5 次。

使用工具 Regex Tester 测试正则表达式示例（184），结果如图 2.39 所示。

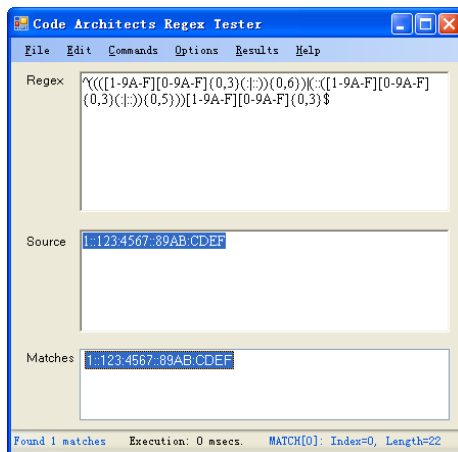


图 2.39 测试正则表达式示例（184）

4. 验证形如::192.168.0.1 的 IP 地址

形如::192.168.0.1 的 IP 地址是以双冒号开头，后接一个 32 位 IP 地址。因此，要验证该类型的 IP 地址，首先要验证 32 位的 IP 地址。前面小节中已经详细地介绍了 32 位 IP 地址的验证方法。以下正则表达式能够验证 32 位的 IP 地址。

```
^(((25[0-5])|(2[0-4]\d)|([01]?\d\d?))\.){3}((25[0-5])|(2[0-4]\d)|([01]?\d\d?))$ (185)
```

综上所述，以下正则表达式能够验证形如::192.168.0.1 的 IP 地址。

```
^::(((25[0-5])|(2[0-4]\d)|([01]?\d\d?))\.){3}((25[0-5])|(2[0-4]\d)|([01]?\d\d?))$ (186)
```

使用工具 Regex Tester 测试正则表达式`^::(((25[0-5])|(2[0-4]\d)|([01]?\d\d?))\.){3}((25[0-5])|`

(2[0-4]\d)([01]?\d\d?))\$, 结果如图 2.40 所示。

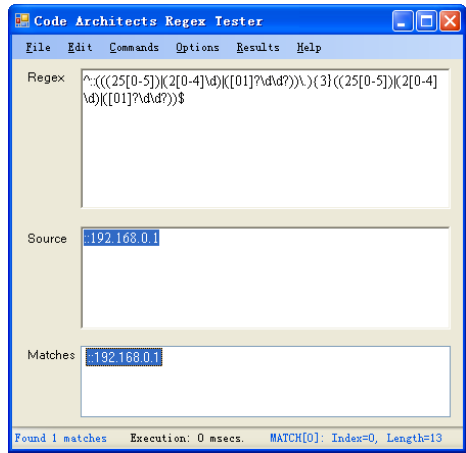


图 2.40 测试正则表达式`^:(((25[0-5])|(2[0-4]\d)|([01]?\d\d?))\.){3}((25[0-5])|(2[0-4]\d)|([01]?\d\d?))$`

第 3 章 常用字符串验证方法

本章的主要内容是介绍常用字符串的验证方法，如英文单词的验证、非单词字符串的验证、文件名称的验证和网络常用元素的验证等。

注意：本章中被验证的字符串可能包含英文单词字符、数字字符和特殊字符。其中，特殊字符是指除英文单词字符和数字字符之外的字符，如/、\、|、,和:等。

3.1 5 种英文单词验证

本节介绍与英文单词相关的验证，如小写英文单词验证、大写英文单词验证、英文单词分隔符验证、不包含验证以及具有重复特征的英文单词验证。

3.1.1 小写英文单词验证

小写英文单词由 26 个小写英文字母（a~z）组成，且英文单词的最小长度为 1。以下正则表达式都能够验证小写英文单词。

<code>[a-z]+</code>	(1)
<code>[a-z]{1,}</code>	(2)

上述两个正则表达式没有验证英文单词的边界。如果给定的字符串为“abc*()”，那么它将匹配字符串“abc”。为了验证一个完整的英文单词，可以使用元字符**b**指定英文单词的边界。以下正则表达式都能够验证小写英文单词，同时指定英文单词的边界。

<code>\b[a-z]+\b</code>	(3)
<code>\b[a-z]{1,}\b</code>	(4)

另外，除了元字符之外，还可以使用指定字符串边界的元字符^和\$来指定英文单词的边界。以下正则表达式都能够验证小写英文单词，同时指定英文单词的边界。

<code>^[a-z]+\$</code>	(5)
<code>^[a-z]{1,}\$</code>	(6)

1. 指定开头字母的英文单词

正则表达式[\[a-z\]+](#)能够验证以任意字母开头的英文单词。然而，有时往往需要验证指定开头字母的英文单词。以下正则表达式都能够验证以字母 a 开头的小写英文单词。

<code>\ba[a-z]*\b</code>	(7)
<code>\ba[a-z]{0,}\b</code>	(8)

以下正则表达式都能够验证以字母 a~g 开头的小写英文单词。

<code>\b[a-g][a-z]*\b</code>	(9)
<code>\b[a-g][a-z]{0,}\b</code>	(10)

2. 指定结尾的英文单词

除了验证上述指定开头字母的英文单词外，有时还需要验证指定结尾的英文单词。如验证以“ing”结尾的英文单词。以下正则表达式都能够验证以“ing”结尾的小写英文单词。

```
\b[a-z]+(?:ing)\b (11)
\b[a-z]{1,}(?:ing)\b (12)
```

正则表达式解释如下。

- 正则表达式**\b[a-z]+(?:ing)\b**：**\b** 匹配英文单词的边界，即英文单词的开始位置或结束位置；**[a-z]+**匹配最小长度为 1、由小写英文字母组成的字符串；**(?:ing)**匹配字符串“ing”，并指定字符串“ing”在英文单词的结尾部分。
- 正则表达式**\b[a-z]{1,}(?:ing)\b**：**\b** 匹配英文单词的边界，即英文单词的开始位置或结束位置；**[a-z]{1,}**匹配最小长度为 1、由小写英文字母组成的字符串；**(?:ing)**匹配字符串“ing”，并指定字符串“ing”在英文单词的结尾部分。

使用工具 **Regex Tester** 分别测试正则表达式**\b[a-z]+(?:ing)\b** 和**\b[a-z]{1,}(?:ing)\b**，结果分别如图 3.1 和图 3.2 所示。

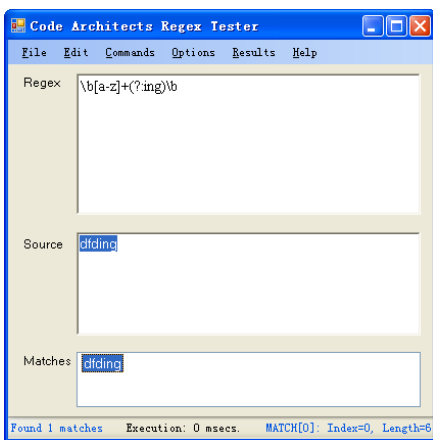


图 3.1 测试正则表达式**\b[a-z]+(?:ing)\b**

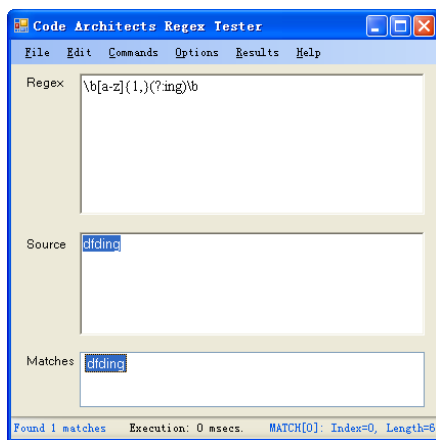


图 3.2 测试正则表达式**\b[a-z]{1,}(?:ing)\b**

3.1.2 大写英文单词验证

大写英文单词由 26 个大写英文字母（A~Z）组成，且英文单词的最小长度为 1。以下正则表达式都能够验证大写英文单词。

```
[A-Z]+ (13)
```

```
[A-Z]{1,} (14)
```

上述两个正则表达式没有验证英文单词的边界。如果给定的字符串为“DEF|*(”，则它将匹配字符串“DEF”。为了验证一个完整的英文单词，可以使用元字符**\b**指定英文单词的边界。以下正则表达式都能够验证大写英文单词，同时指定英文单词边界。

```
\b[A-Z]+\b (15)
```

```
\b[A-Z]{1,}\b (16)
```

另外，除了元字符之外，还可以使用指定字符串边界的元字符**^**和**\$**来指定英文单词边界。以下正则表达式都能够验证大写英文单词，同时指定英文单词边界。

```
^[A-Z]+$ (17)
```

```
^[A-Z]{1,}$ (18)
```

1. 指定开头字母的英文单词

正则表达式**[A-Z]+**能够验证以任意字母开头的英文单词。然而，有时往往需要验证指定开头字母的英文单词。以下正则表达式都能够验证以字母 J 开头的大写英文单词。

```
\bJ[A-Z]*\b (19)
\bJ[A-Z]{0,}\b (20)
```

以下正则表达式都能够验证以字母 C~H 开头的大写英文单词。

```
\b[C-H][A-Z]*\b (21)
\b[C-H][A-Z]{0,}\b (22)
```

2. 指定长度的英文单词

有时往往需要验证指定长度的英文单词，以下正则表达式能够验证长度为 6 的大写英文单词。

```
\b[A-Z]{6}\b (23)
```

3. 指定结尾的英文单词

除了验证上述指定开头字母的英文单词外，有时还需要验证指定结尾的英文单词。如验证以“ED”结尾的英文单词。以下正则表达式都能够验证以“ED”结尾的大写英文单词。

```
\b[A-Z]+(?:ED)\b (24)
\b[A-Z]{1,}(?:ED)\b (25)
```

正则表达式解释如下。

- ❑ 正则表达式**\b[A-Z]+(?:ED)\b**：**\b** 匹配英文单词的边界，即英文单词的开始位置或结束位置；**[A-Z]+** 匹配最小长度为 1、由大写英文字母组成的字符串；**(?:ED)** 匹配字符串“ED”，并指定字符串“ED”出现在英文单词的结尾部分。
- ❑ 正则表达式**\b[A-Z]{1,}(?:ED)\b**：**\b** 匹配英文单词的边界，即英文单词的开始位置或结束位置；**[A-Z]{1,}** 匹配最小长度为 1、由大写英文字母组成的字符串；**(?:ED)** 匹配字符串“ED”，并指定字符串“ED”出现在英文单词的结尾部分。

使用工具 Regex Tester 分别测试正则表达式**\b[A-Z]+(?:ED)\b** 和 **\b[A-Z]{1,}(?:ED)\b**，结果分别如图 3.3 和图 3.4 所示。

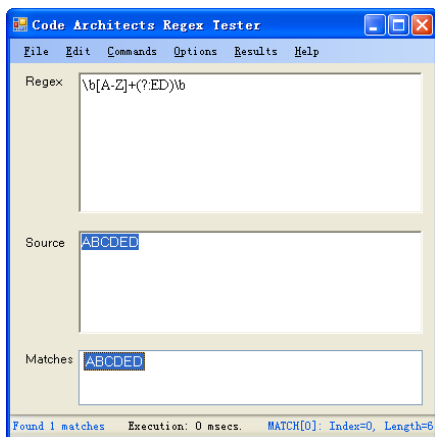


图 3.3 测试正则表达式**\b[A-Z]+(?:ED)\b**

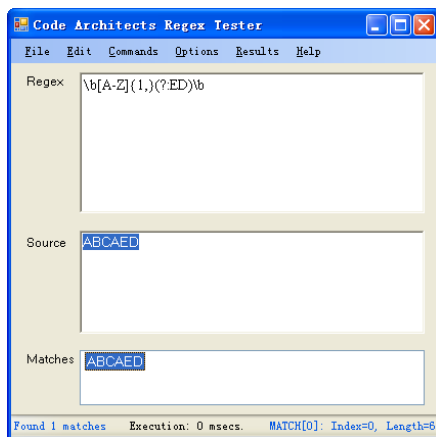


图 3.4 测试正则表达式**\b[A-Z]{1,}(?:ED)\b**

3.1.3 英文单词的分隔符验证

在英文文本中，各个英文单词被分隔符所分开。这些分隔符包括英文标点符号、空白字符等。其中，英文标点符号比较多，如，（逗号）、.（点号）、?（问号）、:（冒号）、;（分号）、'（单引号）、!（感叹号）、"（双引号）、-（连接号）、--（破折号）、...（省略号）、()（小括号）、[]（中括号）、{}（大括号）、`（所有格符号）等。

在英文文本中,有时需要验证两个英文单词是否被标点符号分开,或者是否被指定的分隔符分开。以下正则表达式能够验证英文单词。

```
[a-zA-Z]+ (26)
```

以下正则表达式能够验证英文单词之间的分隔符。

```
[-,.;'!"|(-{2})|(\.{3})|(\(|\)|(\[|])|(\{|}) (27)
```

示例(27)的讲解如下。

- `[-,.;'!"|]`匹配字符`-`、`,`、`.`、`;`、`'`、`"`、`!`或```。
- `-{2}`匹配破折号`--`。
- `\.{3}`匹配省略号`...`。
- `\(|\)|`匹配小括号`()`。
- `\[|]`匹配中括号`[]`。
- `\{|}`匹配大括号`{}`。

使用工具 **Regex Tester** 测试正则表达式`[-,.;'!"|(-{2})|(\.{3})|(\(|\)|(\[|])|(\{|})`, 结果如图 3.5 所示。

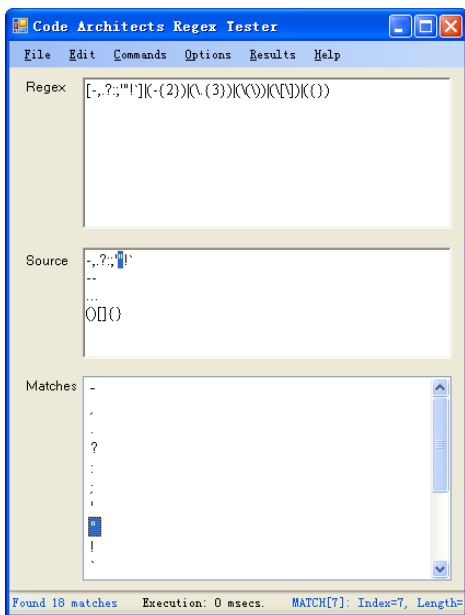


图 3.5 测试正则表达式`[-,.;'!"|(-{2})|(\.{3})|(\(|\)|(\[|])|(\{|)`

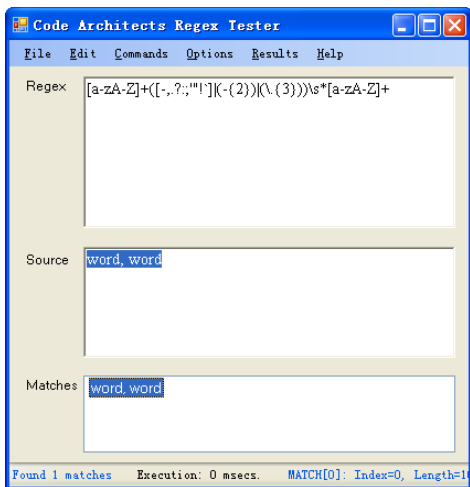
以下正则表达式能够验证英文单词之间的分隔符,且分隔符的两边都是英文单词。

```
[a-zA-Z]+([-,.;'!"|(-{2})|(\.{3}))\s*[a-zA-Z]+ (28)
```

正则表达式解释如下。

- 正则表达式`[a-zA-Z]+([-,.;'!"|(-{2})|(\.{3}))\s*[a-zA-Z]+`: `[a-zA-Z]+`匹配英文单词;
`[-,.;'!"|]`可以匹配`-`、`,`、`.`、`;`、`'`、`"`、`!`、```; `-{2}`匹配破折号`--`; `\.{3}`匹配省略号`...`; `\s*`可以匹配 0 个或多个空白字符。

使用工具 **Regex Tester** 测试正则表达式`[a-zA-Z]+([-,.;'!"|(-{2})|(\.{3}))\s*[a-zA-Z]+`, 结果如图 3.6 所示。

图 3.6 测试正则表达式[\[a-zA-Z\]+\(\[-,?;\"'!\]|\(-{2}\)|\(\{3}\)\)\s*\[a-zA-Z\]+](#)

3.1.4 否定验证

不包含验证其实是一种否定验证。在此，可用它来验证英文单词中不存在指定的字母或者字符串。如验证英文单词中不存在字母 A、验证英文单词中字母 A 之后不能为字母 B 等。

注意：本节所介绍的英文单词均由大写英文字母构成，不考虑小写英文字母。

1. 不包含指定字母的验证

在此，不妨设被指定的不包含的字母为 A，则以下正则表达式都能够验证不包含字母 A 的任意英文单词。

```
\b[B-Z]+\b (29)
```

```
\b[B-Z]{1,}\b (30)
```

若被指定的不包含的字母为 H，则以下正则表达式都能够验证不包含字母 H 的任意英文单词。

```
\b[A-GI-Z]+\b (31)
```

```
\b[A-GI-Z]{1,}\b (32)
```

不包含指定字母的验证实际上是从由字母表 (A~Z) 组成的字符类中排除被指定的不包含的字母。如果被指定的不包含的字母为 C、G、I，那么排除字母 C、G、I 的字符类为 “[A-BD-FHJ-Z]”。因此，以下正则表达式都能够验证不包含字母 C、G、I 的任意英文单词。

```
\b[A-BD-FHJ-Z]+\b (33)
```

```
\b[A-BD-FHJ-Z]{1,}\b (34)
```

2. 字母 A 之后不能为字母 B 类型的英文单词验证

该类型的验证也是一种否定验证，它是指验证英文单词中的某一个字母之后不能为另外一个指定的字母。在此，不妨设字母 B 之后不能为字母 P，则以下正则表达式都能够简单验证字母 B 之后不能为字母 P 的任意英文单词。

```
\b[A-Z]*B[^P][A-Z]*\b (35)
```

```
\b[A-Z]{0,}B[^P][A-Z]{0,}\b (36)
```

3. 示例 (35) 讲解

示例 (35) 的讲解如下。

□ \b 匹配英文单词的边界，即英文单词的开始位置或结束位置。

- ❑ `[A-Z]*`可以匹配空字符串，或者匹配最小长度为 1、由大写英文字母组成的字符串。
- ❑ `B` 匹配字母 B。
- ❑ `[^P]`匹配除字母 P 之外的任意字符。

4. 示例（36）讲解

示例（36）的讲解如下。

- ❑ `\b` 匹配英文单词的边界，即英文单词的开始位置或结束位置。
- ❑ `[A-Z]{0,}`可以匹配空字符串，或者匹配最小长度为 1、由大写英文字母组成的字符串。
- ❑ `B` 匹配字母 B。
- ❑ `[^P]`匹配除字母 P 之外的任意字符。

使用工具 Regex Tester 分别测试正则表达式 `\b[A-Z]*B[^P][A-Z]*\b` 和 `\b[A-Z]{0,}B[^P][A-Z]{0,}\b`，结果分别如图 3.7 和图 3.8 所示。

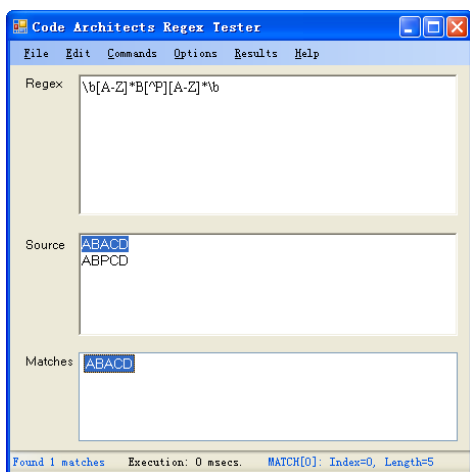


图 3.7 测试正则表达式 `\b[A-Z]*B[^P][A-Z]*\b`

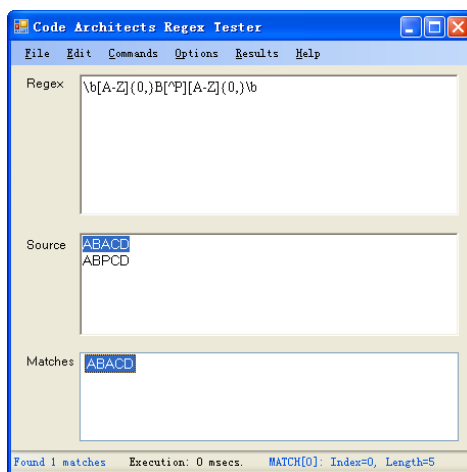


图 3.8 测试正则表达式 `\b[A-Z]{0,}B[^P][A-Z]{0,}\b`

当英文单词的结尾为字母 B 时，正则表达式 `\b[A-Z]*B[^P][A-Z]*\b` 将匹配失败，原因如下。

- ❑ 表达式 `[^P]`总是要匹配一个字符。
- ❑ 当英文单词的结尾为字母 B 时，`[^P]`将会匹配该英文单词之后的分隔符，如空格、句号等。

为了解决上述问题，可以使用零宽度负预测先行断言 (`?!expression`)。它能够断言自身位置的后面不能匹配字符串 `expression`。以下正则表达式都能够简单验证字母 B 之后不能为字母 P 的任意英文单词。

```
\b[A-Z]*B(?![P])[A-Z]*\b (37)
```

```
\b[A-Z]{0,}B(?![P])[A-Z]{0,}\b (38)
```

5. 示例（37）讲解

示例（37）的讲解如下。

- ❑ `\b` 匹配英文单词的边界，即英文单词的开始位置或结束位置。
- ❑ `[A-Z]*`可以匹配空字符串，或者匹配最小长度为 1、由大写英文字母组成的字符串。
- ❑ `B` 匹配字母 B。
- ❑ `(?!P)`能够断言字母 B 之后不能为字母 P。

6. 示例（38）讲解

示例（38）的讲解如下。

- \b 匹配英文单词的边界，即英文单词的开始位置或结束位置。
- [A-Z]{0,} 可以匹配空字符串，或者匹配最小长度为 1、由大写英文字母组成的字符串。
- B 匹配字母 B。
- (?!P) 能够断言字母 B 之后不能为字母 P。

使用工具 Regex Tester 分别测试正则表达式 `\b[A-Z]*B(?:P)[A-Z]*\b` 和 `\b[A-Z]{0,}B(?:P)[A-Z]{0,}\b`，结果分别如图 3.9 和图 3.10 所示。

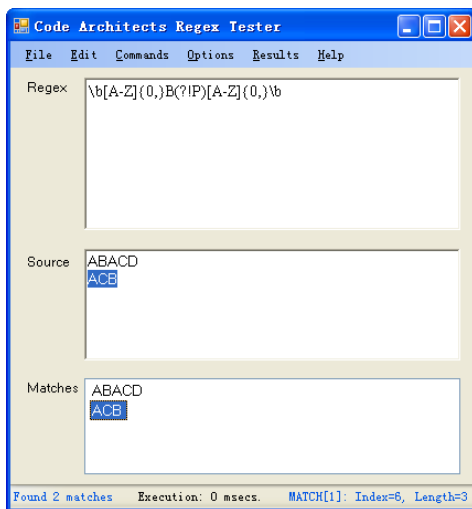
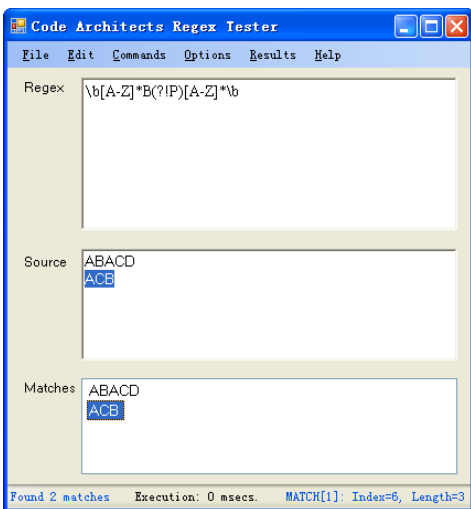


图 3.9 测试正则表达式 `\b[A-Z]*B(?:P)[A-Z]*\b` 图 3.10 测试正则表达式 `\b[A-Z]{0,}B(?:P)[A-Z]{0,}\b`

注意：正则表达式 `\b[A-Z]*B[^P][A-Z]*\b`、`\b[A-Z]{0,}B[^P][A-Z]{0,}\b`、`\b[A-Z]*B(?:P)[A-Z]*\b` 和 `\b[A-Z]{0,}B(?:P)[A-Z]{0,}\b` 都不能精确匹配字母 B 之后不能为字母 P 类型的英文单词。

其中，正则表达式 `\b[A-Z]*B[^P][A-Z]*\b` 和 `\b[A-Z]{0,}B[^P][A-Z]{0,}\b` 除了匹配字母 B 之后不能为字母 P 类型的英文单词外，还能够匹配以下两种情况。

（1）英文单词中包含多个字母 B。此时，上述正则表达式可以匹配字母 B 之后是字母 P 的英文单词。

（2）英文单词字母 B 之后的字符不是字母，而是如字符/或数字等。此时，上述正则表达式可以匹配不是一个英文单词的字符串。

其中，正则表达式 `\b[A-Z]*B(?:P)[A-Z]*\b` 和 `\b[A-Z]{0,}B(?:P)[A-Z]{0,}\b` 除了能够匹配字母 B 之后不为字母 P 类型的英文单词外，还能够匹配上述的第一种情况。

使用工具 Regex Tester 分别对上述两种情况进行测试，结果分别如图 3.11 和图 3.12 所示。其中，测试的正则表达式为 `\b[A-Z]*B[^P][A-Z]*\b`。

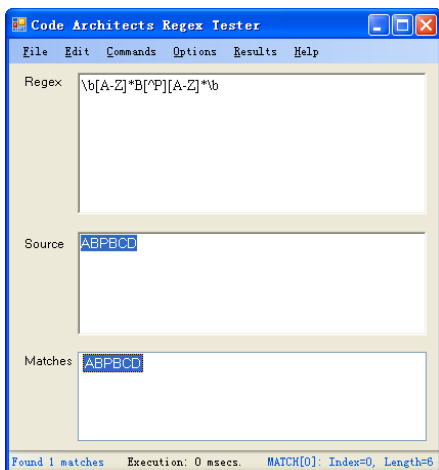


图 3.11 测试 (1) 情况

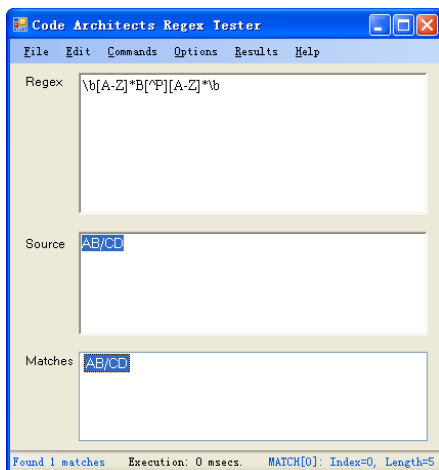


图 3.12 测试 (2) 情况

3.1.5 具有重复特征的英文单词验证

具有重复特征的英文单词是指英文单词中包含连续相同的字母，如 AACCESS、COMMENT 等。因为具有重复特征的英文单词中存在相同字母，因此验证该英文单词时需要使用后向引用。

注意：本节所介绍的英文单词均由大写英文字母构成，不考虑小写英文字母。

1. 以两个相同字母开头的英文单词的验证

以两个相同字母开头的英文单词具有重复特征，可以使用后向引用来验证该类型的英文单词。以下正则表达式都能够验证以两个相同字母开头的英文单词。

```
\b(?:<char>[A-Z])\k<char>[A-Z]*\b (39)
```

```
\b([A-Z])\1[A-Z]*\b (40)
```

2. 示例 (39) 讲解

示例 (39) 的讲解如下。

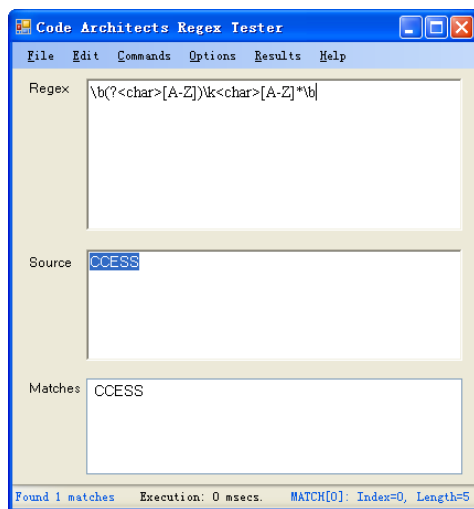
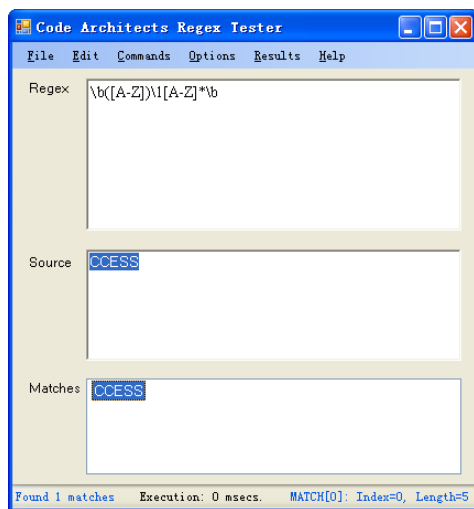
- \b 匹配英文单词的边界，即英文单词的开始位置或结束位置。
- [A-Z]* 可以匹配空字符串，或者匹配最小长度为 1、由大写英文字母组成的字符串。
- 分组(?:<char>[A-Z])将保存匹配内容，并命名为“char”。
- \k<char>使用名称为“char”的分组，它匹配的内容和分组“char”的内容相同。

3. 示例 (40) 讲解

示例 (40) 的讲解如下。

- \b 匹配英文单词的边界，即英文单词的开始位置或结束位置。
- [A-Z]* 可以匹配空字符串，或者匹配最小长度为 1、由大写英文字母组成的字符串。
- 分组([A-Z])将保存匹配内容，并使用默认名称“\1”。
- \1 使用名称为“\1”的分组，它匹配的内容和分组“\1”的内容相同。

使用工具 Regex Tester 分别测试正则表达式 `\b(?:<char>[A-Z])\k<char>[A-Z]*\b` 和 `\b([A-Z])\1[A-Z]*\b`，结果分别如图 3.13 和图 3.14 所示。

图 3.13 测试正则表达式 `\b(?<char>[A-Z])\k<char>[A-Z]*\b`图 3.14 测试正则表达式 `\b([A-Z])\1[A-Z]*\b`

4. 至少存在两个相同字母的英文单词的验证

至少存在两个相同字母的英文单词也具有重复特征，可以使用后向引用来验证该类型的英文单词。以下正则表达式都能够验证至少存在两个相同字母的英文单词。

```
\b[A-Z]* (?<char>[A-Z]) [A-Z]* (\k<char>)+ [A-Z]* \b (41)
```

```
\b[A-Z]* ([A-Z]) [A-Z]* (\1)+ [A-Z]* \b (42)
```

5. 示例（41）讲解

示例（41）的讲解如下。

- `\b` 匹配英文单词的边界，即英文单词的开始位置或结束位置。
- `[A-Z]*` 可以匹配空字符串，或者匹配最小长度为 1、由大写英文字母组成的字符串。
- 分组 `(?<char>[A-Z])` 将保存匹配内容，并命名为“char”。
- `\k<char>` 使用名称为“char”的分组，它匹配的内容和分组“char”的内容相同。
- `(\k<char>)+` 将分组“char”的内容至少重复 1 次。

6. 示例（42）讲解

示例（42）的讲解如下。

- `\b` 匹配英文单词的边界，即英文单词的开始位置或结束位置。
- `[A-Z]*` 可以匹配空字符串，或者匹配最小长度为 1、由大写英文字母组成的字符串。
- 分组 `([A-Z])` 将保存匹配内容，并使用默认名称“`\1`”。
- `\1` 使用名称为“`\1`”的分组，它匹配的内容和分组“`\1`”的内容相同。
- `(\1)+` 将分组“`\1`”的内容至少重复 1 次。

使用工具 Regex Tester 分别测试正则表达式示例（41）和示例（42），结果分别如图 3.15 和图 3.16 所示。

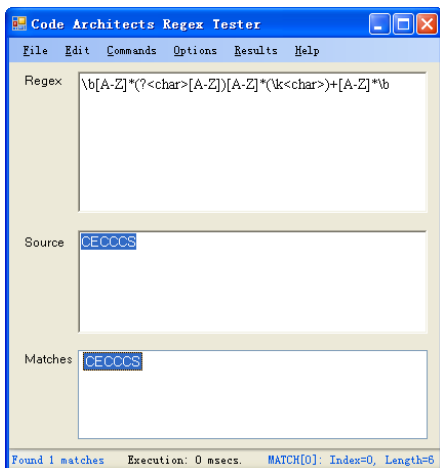


图 3.15 测试正则表达式示例 (41)

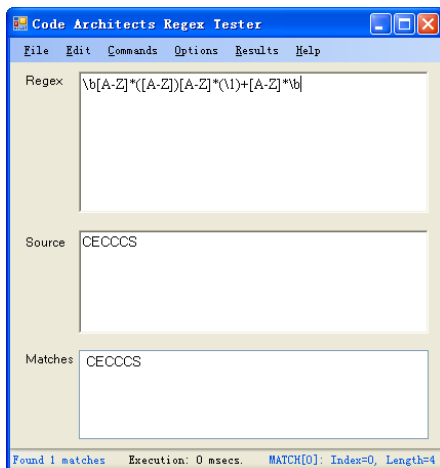


图 3.16 测试正则表达式示例 (42)

3.2 6 种非单词字符串验证

本节内容介绍非单词字符串的验证，如英文标点符号验证、中文标点符号验证、中文文本验证、特殊字符验证、密码验证，以及字符表的分类。

3.2.1 英文标点符号验证

英文标点符号比较多，如，（逗号）、.（点号）、?（问号）、:（冒号）、;（分号）、'（单引号）、!（感叹号）、"（双引号）、-（连接号）、--（破折号）、...（省略号）、()（小括号）、[]（中括号）、{}（大括号）、`（所有格符号）等。以下正则表达式能够验证英文标点符号。

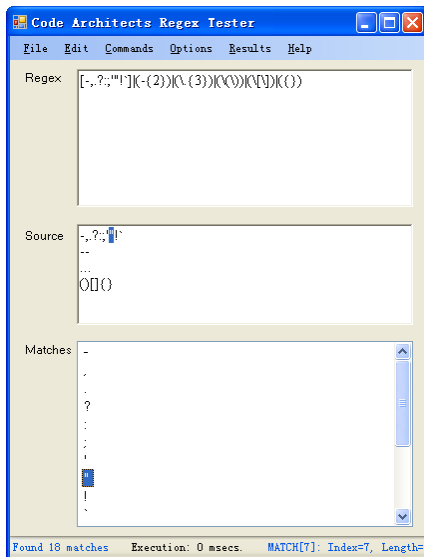
`[-, .?:;'"!]|(-{2})|(\.{3})|(\(\))|(\[\])|(\{ })` (43)

1. 示例 (43) 讲解

示例 (43) 的讲解如下。

- `[-, .?:;'"!]` 可以匹配 `-、,、.、?、:、'、"、!`。
- `-{2}` 匹配破折号 `--`。
- `\{3}` 匹配省略号 `...`。
- `\()` 匹配小括号 `()`。
- `\[\]` 匹配中括号 `[]`。
- `\{ }` 匹配大括号 `{ }`。

使用工具 **Regex Tester** 测试正则表达式 `[-, .?:;'"!]|(-{2})|(\.{3})|(\(\))|(\[\])|(\{ })`，结果如图 3.17 所示。

图 3.17 测试正则表达式 `[-, .?:;'"!]|(-{2})|(\.{3})|(\(\))|(\[\])|(\{ })`

3.2.2 中文标点符号验证

中文标点符号比较多，如，（逗号）、。（句号）、？（问号）、：（冒号）、；（分号）、‘’（单引号）、！（感叹号）、“”（双引号）、—（连接号）、——（破折号）、……（省略号）、（）（小括号）、【】（中括号）、{}（大括号）、、（顿号）、《》（书名号）等。以下正则表达式能够验证中文标点符号。

```
[,。? : ; ‘ ’ ! “ ” —……、 ] ( ( { } ) ( ( 【 】 ) ( ( { } ) ( ( 《 》 )
```

 (44)

1. 示例（44）讲解

示例（44）的讲解如下。

- [,。? : ; ‘ ’ ! “ ” —……、]匹配，、。、?、:、;、‘、’、!、“”、—、……符号。
- —{2}匹配破折号。
- （）匹配小括号。
- 【】匹配中括号。
- {}匹配大括号。
- 《》匹配书名号。

使用工具 Regex Tester 测试正则表达式[,。? : ; ‘ ’ ! “ ” —……、] (({ }) ((【 】) (({ }) ((《 》)，结果如图 3.18 所示。

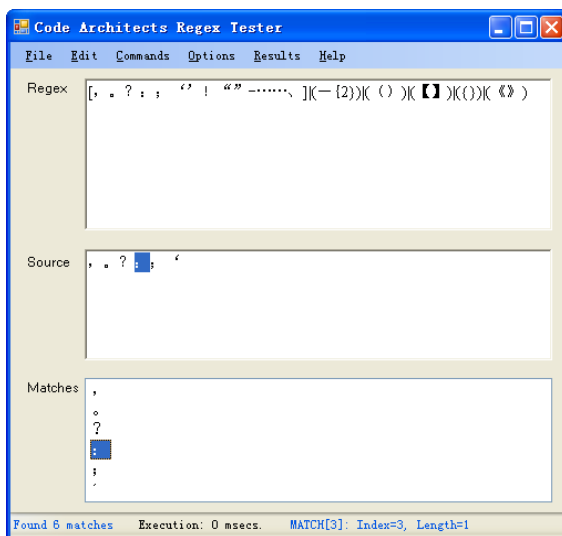


图 3.18 测试正则表达式[,。? : ; ‘ ’ ! “ ” —……、] (({ }) ((【 】) (({ }) ((《 》)

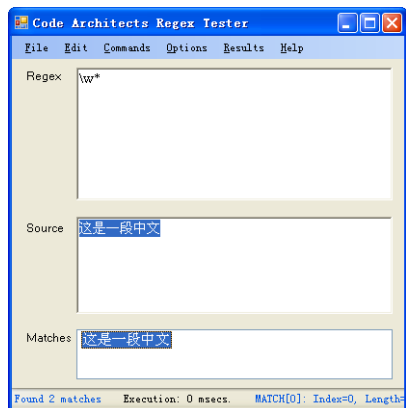
3.2.3 中文文本验证

中文文本的验证比较简单，它可以直接使用元字符\w。以下正则表达式能够验证中文文本。

```
\w*
```

 (45)

使用工具 Regex Tester 测试正则表达式\w*，结果如图 3.19 所示。

图 3.19 测试正则表达式 w^*

3.2.4 特殊字符验证

在此，定义特殊字符为除了数字和字母之外的字符，具体包括 `、-、=、\、[、]、;、'、,、.、/、~、!、@、#、\$、%、^、&、*、(、)、_、+、|、?、>、<、"、:、{和}` 字符。以下正则表达式能够验证一个特殊字符。

```
[ -`=\\[\];',./~!@#$$%^&*()_+|{}:"<>?]
```

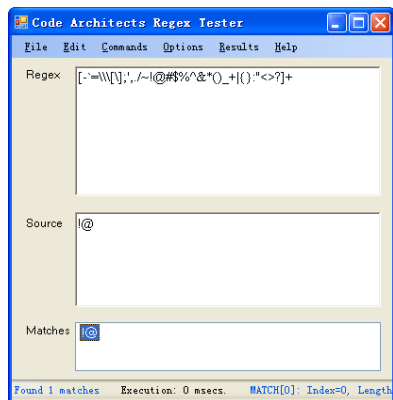
 (46)

以下正则表达式能够验证长度至少为 1、由特殊字符组成的字符串。

```
[ -`=\\[\];',./~!@#$$%^&*()_+|{}:"<>?]+
```

 (47)

使用工具 Regex Tester 测试正则表达式 `[-`=\\[\];',./~!@#$$%^&*()_+|{}:"<>?]+`，结果如图 3.20 所示。

图 3.20 测试正则表达式 `[-`=\\[\];',./~!@#$$%^&*()_+|{}:"<>?]+`

3.2.5 密码验证

当用户登录网站时，一般要求用户提供登录该网站的用户名称和密码。从某种意义上说，密码的复杂程度越高，用户的信息安全性越高。因此，很多网站在注册用户信息或修改用户密码时，往往会提供一个密码强度的说明，提醒用户使用高强度的密码。

目前来说，密码字符可以分为三种：数字、字母和特殊字符（即除数字、字母之外的字符）。用户密码包含上述三种字符中的种类越多，密码的安全性越高。若用户密码只含其中一种则安全性最低，如 123455、abcdef 等。其次是含上述两种字符的密码，如 123abc、123!@#等。密码强

度最好的是包含上述三种字符，如 123QAZ!@#。

注意：本节中的特殊字符包括 `、-、=、\、[、]、;、'、,、.、/、~、!、@、#、\$、%、^、&、*、(、)、_、+、|、?、>、<、"、:、{和} 字符。

1. 只包含数字的密码验证

如果密码中只包含数字，那么该密码是非常简单的。当然，它的安全性也相对较差。以下正则表达式能够验证只包含数字的密码。

```
\d+ (48)
```

该类型密码的安全性随着密码的长度增加而增加。密码越长，它的安全性越高。以下正则表达式能够验证长度至少为 6 位、只包含数字的密码。

```
\d{6,} (49)
```

2. 只包含字母的密码验证

如果密码中只包含字母，那么该密码是非常简单的。当然，它的安全性也相对较差。以下正则表达式能够验证只包含字母的密码。

```
[a-zA-Z]+ (50)
```

该类型密码的安全性随着密码长度的增加而增加。密码越长，它的安全性越高。以下正则表达式能够验证长度至少为 6 位、只包含字母的密码。

```
[a-zA-Z]{6,} (51)
```

3. 只包含特殊字符的密码验证

如果密码中只包含特殊字符，那么该密码是非常简单的。当然，它的安全性也相对较差。以下正则表达式能够验证只包含特殊字符的密码。

```
[-'\=\[\];',./~!@#$$%^&*()_+|{}:"<>?]+ (52)
```

正则表达式解释如下。

正则表达式 `[-'\=\[\];',./~!@#$$%^&*()_+|{}:"<>?]+`：\ 表示字符 \；[表示字符 [；] 表示字符]；其他的每一个字符就表示字符本身。

该类型密码的安全性随着密码长度的增加而增加。密码越长，它的安全性越高。以下正则表达式能够验证长度至少为 6 位、只包含特殊字符的密码。

```
[-'\=\[\];',./~!@#$$%^&*()_+|{}:"<>?]{6,} (53)
```

使用工具 **Regex Tester** 测试正则表达式 `[-'\=\[\];',./~!@#$$%^&*()_+|{}:"<>?]{6,}`，结果如图 3.21 所示。

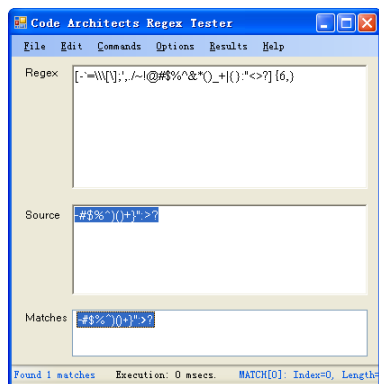


图 3.21 测试正则表达式 `[-'\=\[\];',./~!@#$$%^&*()_+|{}:"<>?]{6,}`

4. 只包含数字和字母的密码验证

如果密码只包含数字和字母，那么该密码的强度是中等强度。当然，它的安全性一般。以下正则表达式能够验证只包含数字和字母的密码。

```
[\\da-zA-Z]*\\d+[a-zA-Z]+[\\da-zA-Z]* (54)
```

正则表达式解释如下。

- 正则表达式`[\\da-zA-Z]*\\d+[a-zA-Z]+[\\da-zA-Z]*`：`\\d` 匹配长度至少为 1 位、由数字组成的字符串；`[a-zA-Z]+` 匹配长度至少为 1 位、由字母组成的字符串；`[\\da-zA-Z]*` 匹配空字符串，或者长度至少为 1 位、由数字或字母组成的字符串。
- `\\d+[a-zA-Z]+` 保证密码既包含了数字又包含特殊字符。

使用工具 Regex Tester 测试正则表达式`[\\da-zA-Z]*\\d+[a-zA-Z]+[\\da-zA-Z]*`，结果如图 3.22 所示。

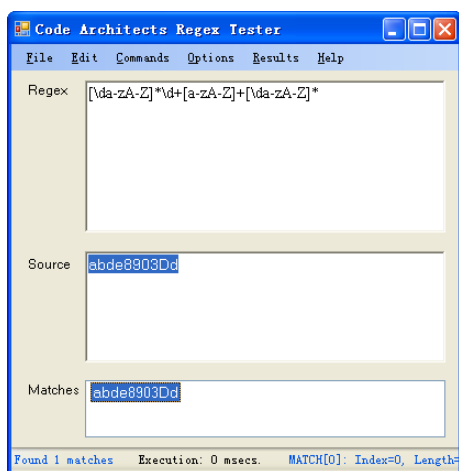


图 3.22 测试正则表达式`[\\da-zA-Z]*\\d+[a-zA-Z]+[\\da-zA-Z]*`

5. 只包含数字和特殊字符的密码验证

如果密码只包含数字和特殊字符，那么该密码的强度是中等强度。当然，它的安全性一般。以下正则表达式能够验证只包含数字和特殊字符的密码。

```
[^-\\d`=\\[\\];',./~!@#%$^&*()_+|{}:"<>?]*\\d+[^`=\\[\\];',./~!@#%$^&*()_+|{}:"<>?]* (55)
```

6. 示例（55）讲解

示例（55）的讲解如下。

- 字符类`[^-\\d`=\\[\\];',./~!@#%$^&*()_+|{}:"<>?]+`：`\\d` 表示任意数字；`\\`表示字符`\`；`[`表示字符`[`；`\]`表示字符`]`；其他的每一个字符就表示字符本身。该字符类可以匹配数字或者特殊字符。
- `\\d+` 匹配最小长度为 1 位、由数字组成的字符串。
- `[^-\\d`=\\[\\];',./~!@#%$^&*()_+|{}:"<>?]+` 匹配最小长度为 1 位、由特殊字符组成的字符串。
- `\\d+[^`=\\[\\];',./~!@#%$^&*()_+|{}:"<>?]+` 保证密码既包含数字又包含特殊字符。

使用工具 Regex Tester 测试正则表达式示例（55），结果如图 3.23 所示。

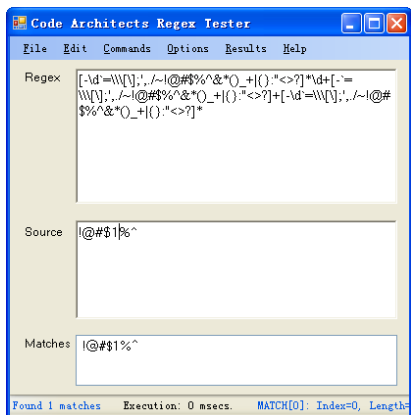


图 3.23 测试正则表达式示例 (55)

7. 只包含字母和特殊字符的密码验证

如果密码只包含字母和特殊字符，那么该密码的强度是中等强度。当然，它的安全性一般。以下正则表达式能够验证只包含字母和特殊字符的密码。

```
[ -a-zA-Z`=\[\] \ ; ' , . / ~ ! @ # $ % ^ & * ( ) _ + | { } : " < > ? ] * [ a-zA-Z ] + [ - ` = \ [ \ ] \ ; ' , . / ~ ! @ # $ % ^ & * ( ) _ + | { } : " < > ? ] * (56)
```

8. 示例 (56) 讲解

示例 (56) 的讲解如下。

- ❑ 字符类 `[-a-zA-Z`=\[\] \ ; ' , . / ~ ! @ # $ % ^ & * () _ + | { } : " < > ?]`：a-z 表示小写字母；A-Z 表示大写字母；\表示字符\；\表示字符[；\表示字符]；其他的每一个字符都表示字符本身。该字符类可以匹配字母或者特殊字符。
- ❑ `[a-zA-Z]`匹配最小长度为 1 位、由字母组成的字符串。
- ❑ `[- ` = \ [\] \ ; ' , . / ~ ! @ # $ % ^ & * () _ + | { } : " < > ?]`匹配最小长度为 1 位、由特殊字符组成的字符串。
- ❑ `[a-zA-Z] + [- ` = \ [\] \ ; ' , . / ~ ! @ # $ % ^ & * () _ + | { } : " < > ?] *`保证密码既包含字母又包含特殊字符。

使用工具 Regex Tester 测试正则表达式示例 (56)，结果如图 3.24 所示。

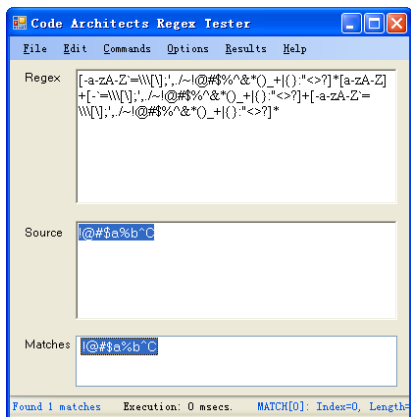


图 3.24 测试正则表达式示例 (56)

9. 只包含数字、字母和特殊字符的密码验证

如果密码只包含数字、字母和特殊字符，那么该密码的强度是高强度。当然，它的安全性相

对比较好。以下正则表达式能够验证只包含数字、字母和特殊字符的密码。

```
[-\da-zA-Z`=\\[\];',./~!@#%&*()_+|{}:"<>?]*
(
  (\d+[\a-zA-Z]+[-`=\\[\];',./~!@#%&*()_+|{}:"<>?]+) #数字开头
  | (\d+[-`=\\[\];',./~!@#%&*()_+|{}:"<>?]+[\a-zA-Z]+) #数字开头
  | ([\a-zA-Z]+\d+[-`=\\[\];',./~!@#%&*()_+|{}:"<>?]+) #字母开头
  | ([\a-zA-Z]+[-`=\\[\];',./~!@#%&*()_+|{}:"<>?]+\d+) #字母开头
  | ([-`=\\[\];',./~!@#%&*()_+|{}:"<>?]+\d+[\a-zA-Z]+) #特殊字符开头
  | ([-`=\\[\];',./~!@#%&*()_+|{}:"<>?]+[\a-zA-Z]+\d+) #特殊字符开头
)
[-\da-zA-Z`=\\[\];',./~!@#%&*()_+|{}:"<>?]*
```

(57)

10. 示例（57）讲解

示例（57）的讲解如下。

- \d+匹配最小长度为 1、由数字组成的字符串。
- [\a-zA-Z]+匹配最小长度为 1、由字母组成的字符串。
- [-`=\\[\];',./~!@#%&*()_+|{}:"<>?]+匹配最小长度为 1、由特殊字符组成的字符串。
- \d+[\a-zA-Z]+[-`=\\[\];',./~!@#%&*()_+|{}:"<>?]+匹配形如“数字+字母+特殊字符”类型的字符串。
- (\d+[-`=\\[\];',./~!@#%&*()_+|{}:"<>?]+[\a-zA-Z]+)匹配形如“数字+特殊字符+字母”类型的字符串。
- [\a-zA-Z]+\d+[-`=\\[\];',./~!@#%&*()_+|{}:"<>?]+匹配形如“字母+数字+特殊字符”类型的字符串。
- [\a-zA-Z]+[-`=\\[\];',./~!@#%&*()_+|{}:"<>?]+\d+匹配形如“字母+特殊字符+数字”类型的字符串。
- [-`=\\[\];',./~!@#%&*()_+|{}:"<>?]+\d+[\a-zA-Z]+匹配形如“特殊字符+数字+字母”类型的字符串。
- [-`=\\[\];',./~!@#%&*()_+|{}:"<>?]+[\a-zA-Z]+\d+匹配形如“特殊字符+字母+数字”类型的字符串。
- 字符类[-\da-zA-Z`=\\[\];',./~!@#%&*()_+|{}:"<>?]*: \d 表示任意数字; a-z 表示小写字母; A-Z 表示大写字母; \\表示字符\; \[表示字符[; \]表示字符]; 其他的每一个字符都表示字符本身。该字符类可以匹配数字、字母或者特殊字符。

使用工具 Regex Tester 测试正则表达式示例（57），结果如图 3.25 所示。



图 3.25 测试正则表达式示例（57）

3.2.6 字符表的分类

在正则表达式中，可以把字符表进行分类，每一类代表不同的字符。字符表的常用分类如表 3-1 所示。

注意：本节中的字符表的分类为 .NET Framework 所支持。

表 3-1 字符表的常用分类

表达式	说明
\p{Ll}	等同于\p{Lowercase_Letter}，小写字母
\p{Lu}	等同于\p{Uppercase_Letter}，大写字母
\p{Lt}	等同于\p{Titlecase_Letter}，标题大小写字母
\p{L&}	等同于\p{Ll}、\p{Lu}和\p{Lt}
\p{Lm}	等同于\p{Modifier_Letter}
\p{Lo}	等同于\p{Other_Letter}，不能被修改的、一般用于固定名称等
\p{Mn}	等同于\p{Non_Spacing_Mark}
\p{Mc}	等同于\p{Spacing_Combining_Mark}
\p{Me}	等同于\p{Enclosing_Mark}
\p{Zs}	等同于\p{Space_Separator}，空白字符
\p{Zl}	等同于\p{Line_Separator}，行分隔字符（U+2028）
\p{Zp}	等同于\p{Paragraph_Separator}，段分隔字符（U+2029）
\p{Sm}	等同于\p{Math_Symbol}，数学运算符
\p{Sc}	等同于\p{Currency_Symbol}，货币符号
\p{Sk}	等同于\p{Modifier_Symbol}
\p{So}	等同于\p{Other_Symbol}
\p{Nd}	等同于\p{Decimal_Digit_Number}，数字字符
\p{Nl}	等同于\p{Letter_Number}，罗马数字字符
\p{No}	等同于\p{Other_Number}，除了上述数字字符之外的数字字符
\p{Pd}	等同于\p{Dash_Punctuation}，破折号标点符号
\p{Ps}	等同于\p{Open_Punctuation}，如《等左双标点符号
\p{Pe}	等同于\p{Close_Punctuation}，如》等右双标点符号
\p{Pi}	等同于\p{Initial_Punctuation}，如<等左单标点符号
\p{Pf}	等同于\p{Final_Punctuation}，如>等右单标点符号
\p{Pc}	等同于\p{Connector_Punctuation}，连接标点符号
\p{Po}	等同于\p{Other_Punctuation}，其他标点符号
\p{Cc}	等同于\p{Control}，控制符号
\p{Cf}	等同于\p{Format}
\p{Co}	等同于\p{Private_Use}
\p{Cn}	等同于\p{Unassigned}

3.3 常用的文件名称和路径验证

每一个文件都拥有它自己的名称，该名称由两部分组成：文件名和文件扩展名。在此，把由文件名和文件扩展名组成的名称称为文件全名。本节将介绍与文件名称和文件路径相关的验证。

3.3.1 通配符

在 Windows 或 DOS 运算系统中，最常用的通配符是*和?。一个字符?可以匹配一个任意字符。

以下表达式可以匹配任意的、长度为 3 的、文件扩展名为 doc 的文件全名。

```
???.doc
```

一个字符*可以匹配任意长度的字符串。以下表达式可以匹配任意名称的、文件扩展名为 doc 的文件全名。

```
*.doc
```

以下表达式可以匹配任意名称的文件全名。

```
*.*
```

注意：Windows 和 DOS 运算系统中的通配符*和?, 与正则表达式中的字符*和?的意义不同, 请读者加以区别。

3.3.2 指定文件扩展名的验证

有时需要验证某一类型的文件, 即验证文件扩展名。在此, 不妨设被验证的文件扩展名为 doc (即 Word 文档)。验证同一类型的文件时, 文件名可以为长度至少为 1 的任意字符串。以下正则表达式能够简单验证文件扩展名为 doc 的文件全名。

```
.\+.doc (58)
```

正则表达式.\+.doc 能够简单验证文件扩展名为 doc 的文件全名。其中, .+匹配长度至少为 1、由非换行字符组成的字符串。如果被验证的字符串为 “_//.doc”, 则正则表达式.\+.doc 能够匹配该字符串。然而, 文件名中不能包含\、/、:、*、?、“”、<、>和|字符, 因此, 正则表达式.\+.doc 只能简单验证文件扩展名为 doc 的文件全名。以下正则表达式能够精确验证文件扩展名为 doc 的文件全名。

```
^[^\\\/:*?"<>|]+\+.doc$ (59)
```

1. 示例 (59) 讲解

示例 (59) 的讲解如下。

- ^和\$分别匹配字符串的开始位置和结束位置。
- 字符类[^\\\/:*?"<>|]将匹配除\、/、:、*、?、“”、<、>、|之外的任意字符。
- [^\\\/:*?"<>|]+匹配长度至少为 1、由除\、/、:、*、?、“”、<、>、|之外的任意字符组成的字符串。
- \匹配点号, doc 匹配文件扩展名。

使用工具 Regex Tester 测试正则表达式^[^\\\/:*?"<>|]+\+.doc\$, 结果如图 3.26 所示。

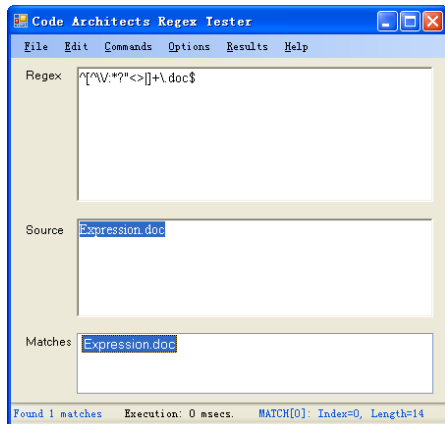


图 3.26 测试正则表达式^[^\\\/:*?"<>|]+\+.doc\$

3.3.3 指定文件名的验证

指定文件名的验证比指定文件扩展名的验证要简单得多。在此，不妨设被验证的文件名为 `filename`。文件扩展名一般由单词字符组成的、长度至少为 1 的字符串。以下正则表达式能够验证文件名为 `filename` 的文件全名。

```
^filename\\.\\w+$ (60)
```

正则表达式解释如下。

正则表达式 `^filename\\.\\w+$`：`^`和`$`分别匹配字符串的开始位置和结束位置；`filename` 匹配文件名；`\\`匹配点号；`\\w+`匹配文件扩展名，它是一个由单词字符组成的、长度至少为 1 的字符串。

使用工具 `Regex Tester` 测试正则表达式 `^filename\\.\\w+$`，结果如图 3.27 所示。

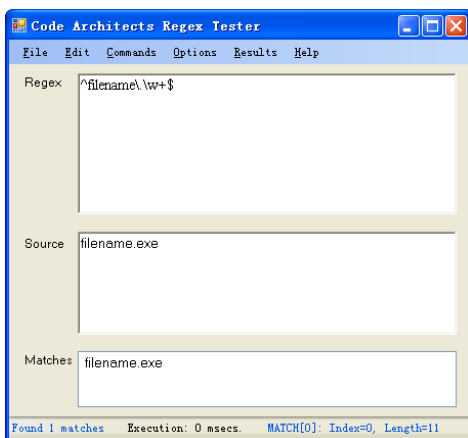


图 3.27 测试正则表达式 `^filename\\.\\w+$`

3.3.4 包含指定字符串的文件全名验证

除了上述指定文件名或文件扩展名的验证之外，还存在一种常用验证：即被验证的文件名中包含给定的字符串。在此，不妨设被包含的字符串为“`ab`”。如果某一个文件名中包含字符串“`ab`”，那么该文件名将通过验证。以下正则表达式能够简单验证包含字符串为“`ab`”的文件全名。

```
^.\\*ab\\.\\*\\.\\w+$ (61)
```

正则表达式解释如下。

- 正则表达式 `^.*ab\\.*\\.\\w+$`：`^`和`$`分别匹配字符串的开始位置和结束位置；`*`可以匹配空字符串，或者匹配长度至少为 1、由非换行字符组成的字符串；`\\`匹配点号；`\\w+`匹配文件扩展名，它是一个由单词字符组成的、长度至少为 1 的字符串。

正则表达式 `^.*ab\\.*\\.\\w+$`能够简单验证包含字符串为“`ab`”的文件全名。如果被验证的字符串为“`_ab<>.doc`”，则正则表达式 `^.*ab\\.*\\.\\w+$`能够匹配该字符串。然而，文件名中却不能够包含`\\`、`/`、`:`、`*`、`?`、`"`、`<`、`>`和`|`字符。因此，正则表达式 `^.*ab\\.*\\.\\w+$`只能够简单验证包含的字符串为“`ab`”的文件全名。以下正则表达式能够精确验证包含字符串为“`ab`”的文件全名。

```
^[^\\w:.*?<>|]*ab[^\w:.*?<>|]*\\.\\w+$ (62)
```

1. 示例（62）讲解

示例（62）的讲解如下。

- `^`和`$`分别匹配字符串的开始位置和结束位置。
- 字符类 `[^\w:.*?<>|]`将匹配除`\\`、`/`、`:`、`*`、`?`、`"`、`<`、`>`、`|`之外的任意字符。

- ❑ `[\W:.*?<>]*`可以匹配空字符串, 或者匹配长度至少为 1、由除\、/、:、*、?、"、<、>和|之外的任意字符组成的字符串。
- ❑ `\.`匹配点号。
- ❑ `\w+`匹配文件扩展名, 它是一个由单词字符组成的、长度至少为 1 的字符串。

使用工具 **Regex Tester** 测试正则表达式`^[^\\:.*?<>]*ab[\\:.*?<>]*\\.\\w+$`, 结果如图 3.28 所示。



图 3.28 测试正则表达式`^[^\\:.*?<>]*ab[\\:.*?<>]*\\.\\w+$`

3.3.5 排除两端存在空白字符的文件全名验证

前面小节中验证文件全名的正则表达式都没有考虑文件名两端存在空白字符的情况。然而, 文件名的两端是不允许存在空白字符的。下面介绍排除两端存在空白字符的文件全名的验证方法。

要验证一个字符串不是以空白字符开头, 可以使用零宽度负预测先行断言, 即正则表达式`(?!expression)`。该断言能够指定此位置的后面不能匹配表达式 `expression`。以下正则表达式能够验证不是以空白字符开头的、长度至少为 1 的字符串。

`^(?!).+$` (63)

正则表达式解释如下。

- ❑ 正则表达式`^(?!).+$`: `^`和`$`分别匹配字符串的开始位置和结束位置; `+`匹配长度至少为 1、由非换行字符组成的字符串; `(?!)`是零宽度负预测先行断言, 它断言字符串不能以空白字符开头。

使用工具 **Regex Tester** 测试正则表达式`^(?!).+$`, 结果如图 3.29 所示。

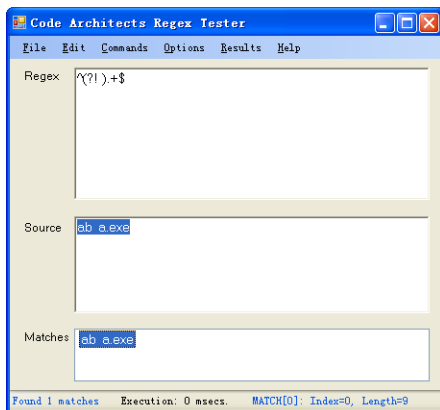


图 3.29 测试正则表达式`^(?!).+$`

如果文件名不能以空白字符结束，实际上是要验证.字符之前不能为空白字符。零宽度负回顾后发断言可以实现该功能，表达式为`(?<!experssion)`，它断言自身位置的前面不能匹配字符串`experssion`。以下正则表达式能够验证.字符之前不能为空白字符的字符串。

```
^.+(?<!\. )\.$ (64)
```

正则表达式解释如下。

- 正则表达式`^.+(?<!\.)\.$`：`^`和`$`分别匹配字符串的开始位置和结束位置；`+`匹配长度至少为 1、由非换行字符组成的字符串；`(?<!\.)`是零宽度负回顾后发断言，它断言字符串的前面不能为空白字符。

使用工具 **Regex Tester** 测试正则表达式`^.+(?<!\.)\.$`，结果如图 3.30 所示。

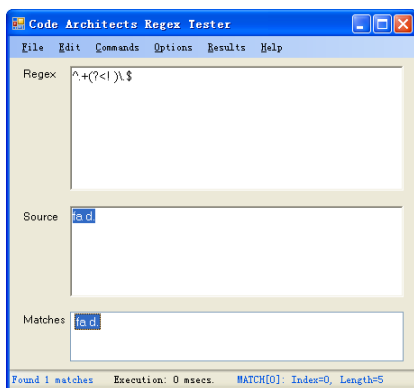


图 3.30 测试正则表达式`^.+(?<!\.)\.$`

综上所述，以下正则表达式能够验证文件名的两端不存在空白字符的文件全名。

```
^(?! ) [^\W/:*?"<>|]+(?! ) \. \w+$ (65)
```

1. 示例（65）讲解

示例（65）的讲解如下。

- `^`和`$`分别匹配字符串的开始位置和结束位置。
- 字符类`[^\W/:*?"<>|]`将匹配除`\`、`/`、`:`、`*`、`?`、`"`、`<`、`>`和`|`之外的任意字符。
- `[^\W/:*?"<>|]`匹配长度至少为 1、由除`\`、`/`、`:`、`*`、`?`、`"`、`<`、`>`和`|`之外的任意字符组成的字符串。
- `\.`匹配点号。
- `\w`匹配文件扩展名，它是一个由单词字符组成、长度至少为 1 的字符串。
- `(?!)`是零宽度负预测先行断言，它断言字符串不能以空白字符开头。
- `(?!)`是零宽度负回顾后发断言，它断言字符串的前面不能为空白字符。

使用工具 **Regex Tester** 测试正则表达式`^(?!) [^\W/:*?"<>|]+(?!) \. \w+$`，结果如图 3.31 所示。

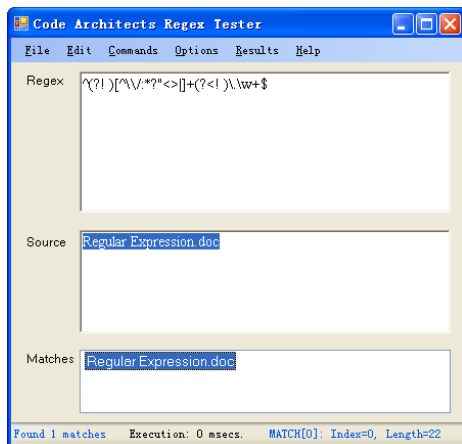


图 3.31 测试正则表达式`^(?!) [^\W/:*?"<>|]+(?!) \. \w+$`

3.3.6 文件路径验证

在 Windows 或 DOS 运算系统中, 文件路径满足以下 5 个特点。

- ❑ 开头字符为硬盘的盘符, 如 C:、D:等。
- ❑ 被字符“\”分割的字符串。
- ❑ 被分割后的每一个字符串要么是盘符, 要么是文件夹名称。其中, 文件夹名称的末尾可以包含空白字符。
- ❑ 如果字符“\”后面为文件夹名称, 则该字符后面不能紧接空白字符。
- ❑ 如果字符“\”前面为盘符, 则该字符后面可以紧接空白字符。

1. 盘符验证

盘符一般由一个英文字符和冒号(:)组成。以下正则表达式能够验证盘符。

```
[a-zA-Z]: (66)
```

2. 文件夹名称验证

在 Windows 或 DOS 运算系统中, 文件夹名称和文件名称命名规则相似。根据前面小节的讲解, 可以知道验证文件夹名称的正则表达式。以下正则表达式能够验证文件夹名称。

```
^(?!)[^\\/:*?"<>|]+(?!)$ (67)
```

3. 文件路径验证

综合上述两点, 可以知道验证文件路径的正则表达式。以下正则表达式能够验证文件路径。

```
^[a-zA-Z]:(((\\(?!)[^\\/:*?"<>|]+\\?)(\\))\\s*$ (68)
```

4. 示例(68)讲解

示例(68)的讲解如下。

- ❑ ^和\$分别匹配字符串的开始位置和结束位置。
- ❑ [a-zA-Z]:匹配盘符。
- ❑ (\\)匹配字符“\”, 它和[a-zA-Z]:\s*组合而成的表达式[a-zA-Z]:\\s*将匹配硬盘根目录的路径。
- ❑ [^\\/:*?"<>|]+匹配文件夹名称。
- ❑ (?!)[^\\/:*?"<>|]+匹配不是以空白字符开头的文件夹名称。
- ❑ \\(?!)[^\\/:*?"<>|]+匹配由 1 个或多个“字符/+不是以空白字符开头的文件夹名称”组成的字符串, 如
\\Book\\Rexpression。
- ❑ \\?可以匹配 0 个或 1 个字符“\”, 它将匹配路径的最后一个字符“\”。
- ❑ \s*匹配路径最后的空白字符。

使用工具 Regex Tester 测试正则表达式`^[a-zA-Z]:(((\\(?!)[^\\/:*?"<>|]+\\?)(\\))\\s*$`, 结果如图 3.32 所示。

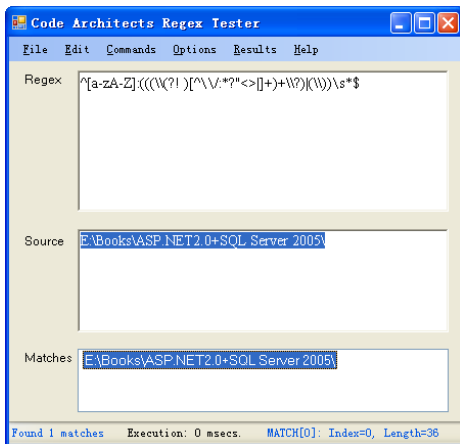


图 3.32 测试正则表达式`^[a-zA-Z]:(((\\(?!)[^\\/:*?"<>|]+\\?)(\\))\\s*$`

3.4 4 种网络常用元素验证

本节介绍网络常用元素验证，如电子邮件验证、主机名称验证、HTTP 地址验证和 FTP 地址验证。

3.4.1 电子邮件验证

电子邮件是当前网络时代最常用的应用之一。邮件地址一般由名称、字符@和域名后缀组成，如 admin@admin.com、123_d@123.com 等。

1. 简单邮件地址验证

如果邮件地址只包含单词字符，则定义该类型的邮件地址为简单邮件地址。以下正则表达式能够验证简单邮件地址。

```
\w+@\w+(\.\w+)+ (69)
```

正则表达式解释如下。

- 正则表达式 `\w+@\w+(\.\w+)+`：`\w+`能够匹配长度至少为 1、由单词字符组成的字符串；`@`匹配邮件地址中的字符`@`；`\.`匹配字符`.`；`(\.\w+)+`能够匹配一个或多个形如“字符`.`由单词字符组成的字符串”的字符串。

使用工具 Regex Tester 测试正则表达式 `\w+@\w+(\.\w+)+`，结果如图 3.33 所示。

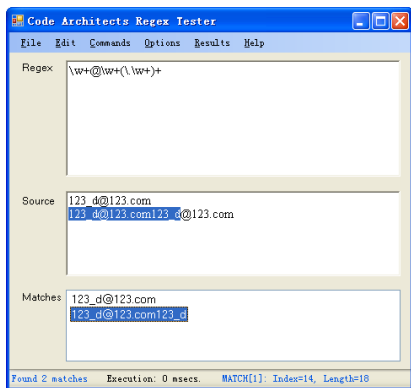


图 3.33 测试正则表达式 `\w+@\w+(\.\w+)+`

2. 扩展邮件地址验证

如果邮件地址不仅包含单词字符，还包含如`-`、`+`、`.`、`'`等字符，则定义该类型的邮件地址为扩展邮件地址。以下正则表达式能够验证扩展邮件地址。

```
\w+([-+.']\w+)*@\w+([-.\w+)*\.\w+([-.\w+)* (70)
```

3. 示例（70）讲解

示例（70）的讲解如下。

- `\w+`能够匹配长度至少为 1、由单词字符组成的字符串。
- `[-+.']\w+`匹配`-`、`+`、`.`、`'`字符；`[-+.']\w+`匹配以`-`、`+`、`.`或`'`字符开头的、后接长度至少为 1 的单词字符串；`([-+.']\w+)*`表示以`-`、`+`、`.`或`'`字符开头的、后接长度至少为 1 的单词字符串可以不出现或者至少出现 1 次。

- @匹配邮件地址中的字符@。
- [-.]匹配-或.字符；[-.]\w+匹配以-或.字符开头的、后接长度至少为 1 的单词字符串；([-.]\w+)*表示以-或.字符开头的、后接长度至少为 1 的单词字符串可以不出现或者至少出现 1 次。
- \.匹配字符.

注意：正则表达式\w+([-.'\w+)*@\w+([-.'\w+)*\.\w+([-.'\w+)*允许邮件地址包含-、+、.和'等字符，如邮件地址 aaa+bb@cc-w.cd.com 能够被验证。

使用工具 Regex Tester 测试正则表达式\w+([-.'\w+)*@\w+([-.'\w+)*\.\w+([-.'\w+)*，结果如图 3.34 所示。

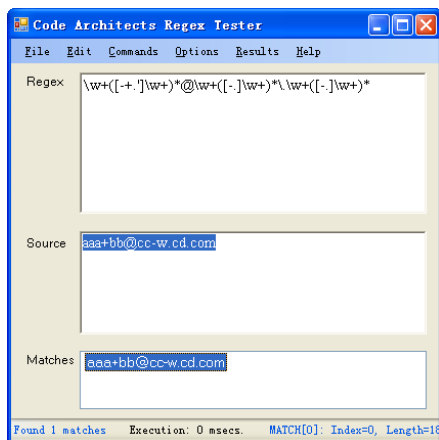


图 3.34 测试正则表达式\w+([-.'\w+)*@\w+([-.'\w+)*\.\w+([-.'\w+)*

3.4.2 主机名称验证

互联网上的主机名称一般由单词字符和字符组成，如 myhost.com、wname.net 或 123.cn 等。

主机名称被字符分割，被分割的每一个字符串由单词字符组成，但是字符不能是该字符串的第一个或最后一个字符。另外，被分割后的每一个字符串的最大长度为 63、最小长度为 1。以下正则表达式能够验证被分割后的每一个字符串。其中，长度为 1 的字符串由表达式[a-zA-Z0-9]匹配，长度大于 1 的字符串由表达式[a-zA-Z0-9]\w{0,61}[a-zA-Z0-9]匹配。

```
(([a-zA-Z0-9]\w{0,61}[a-zA-Z0-9])|([a-zA-Z0-9]))
```

 (71)

正则表达式解释如下。

- [a-zA-Z0-9]能够匹配除字符-之外的任意单词字符。
- \w{0,61}可以匹配由单词字符组成的、最小长度为 0、最大长度为 61 的字符串。

上述正则表达式可以扩展为验证主机名称的正则表达式，以下正则表达式能够简单验证主机名称。

```
((([a-zA-Z0-9]\w{0,61}[a-zA-Z0-9]\.)*|([a-zA-Z0-9]\.))*([a-zA-Z0-9]\w{0,61}[a-zA-Z0-9])|([a-zA-Z0-9])
```

 (72)

正则表达式解释如下。

- [a-zA-Z0-9]能够匹配除字符-之外的任意单词字符。
- \w{0,61}可以匹配由单词字符组成的、最小长度为 0、最大长度为 61 的字符串。
- \.匹配字符.

- `(([a-zA-Z0-9]\w{0,61}[a-zA-Z0-9]\.)([a-zA-Z0-9]\.))*` 可以匹配至少重复“被分割后的每一个字符串+字符.” 0 次的字符串。

使用工具 Regex Tester 测试正则表达式示例（72），结果如图 3.35 所示。

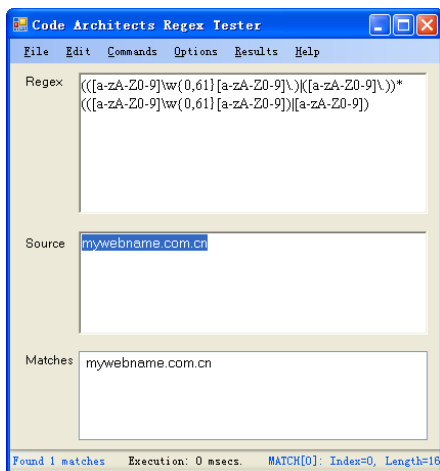


图 3.35 测试正则表达式示例（72）

然而，在目前的网络主机名称中，被分割后的最后一个字符串是固定的，而不是由单词字符串组成的任意字符串，如 `com`、`edu`、`gov`、`millnet`、`org`、`net`、`cn` 等。正则表达式 `(([a-zA-Z0-9]\w{0,61}[a-zA-Z0-9]\.)([a-zA-Z0-9]\.))*` 仅能够简单验证主机名称。以下正则表达式能够精确验证主机名称。

```
(([a-zA-Z0-9]\w{0,61}[a-zA-Z0-9]\.)([a-zA-Z0-9]\.)* (com|edu|gov|int|mil|net|org|biz|info|name|museum|coop|aero|[a-z][a-z])
```

(73)

1. 示例（73）讲解

示例（73）的讲解如下。

- `[a-zA-Z0-9]` 能够匹配除字符 `-` 之外的任意单词字符。
- `\w{0,61}` 可以匹配由单词字符组成的、最小长度为 0、最大长度为 61 的字符串。
- `\.` 匹配字符 `.`。
- `(([a-zA-Z0-9]\w{0,61}[a-zA-Z0-9]\.)([a-zA-Z0-9]\.))*` 可以匹配至少重复“被分割后的每一个字符串+字符.” 0 次的字符串。
- `com|edu|gov|int|mil|net|org|biz|info|name|museum|coop|aero|[a-z][a-z]` 可以匹配给定的字符串（如 `com`、`edu`、`gov`、`millnet`、`org`、`net` 等）或者由字母组成的长度为 2 的字符串。

使用工具 Regex Tester 测试正则表达式示例（73），结果如图 3.36 所示。



图 3.36 测试正则表达式示例（73）

3.4.3 HTTP 地址验证

HTTP 地址一般是以字符串“http://”或“https://”开头的字符串。它可以被字符.、/、?、&、%、=分割。如 http://www.ab.com、https://ac.cn 或 http://cn.net/2007/07/06/aa.aspx?ID=1 等。

HTTP 地址在字符串“http://”或“https://”之后，首先是一个以字符.分割的字符串。以下正则表达式能够验证该字符串。

```
([\w-]+\.)+[\w-]+ (74)
```

1. 示例（74）讲解

示例（74）的讲解如下。

- [\w-]能够匹配单词字符和连接符号-。
- \.匹配字符。
- [\w-]+\.能够匹配由单词字符和连接符号-组成的以字符串开头的、以字符.结尾的字符串。
- ([\w-]+\.)+能够匹配 1 个或多个由单词字符和连接符号-组成的以字符串开头的、以字符.结尾的字符串。

以下正则表达式能够验证 HTTP 地址中除上述字符串之外的字符串。

```
(/[\w- ./?%&=]*)? (75)
```

2. 示例（75）讲解

示例（75）的讲解如下。

- /匹配字符/。
- [\w- ./?%&=]能够匹配单词字符、-、（空格）、.、/、?、%、&和=；[\w- ./?%&=]*能够匹配空字符串，或者由单词字符、-、（空格）、.、/、?、%、&和=组成的长度至少为 1 的字符串。
- ([\w- ./?%&=]*)?表示表达式[\w- ./?%&=]*匹配的字符串可以不出现或者只出现 1 次。

综上所述，以下正则表达式能够验证完整的 HTTP 地址。

```
http(s)?://([\w-]+\.)+[\w-]+(/[\w- ./?%&=]*)? (76)
```

使用工具 Regex Tester 测试正则表达式 http(s)?://([\w-]+\.)+[\w-]+(/[\w- ./?%&=]*)?，结果如图 3.37 所示。

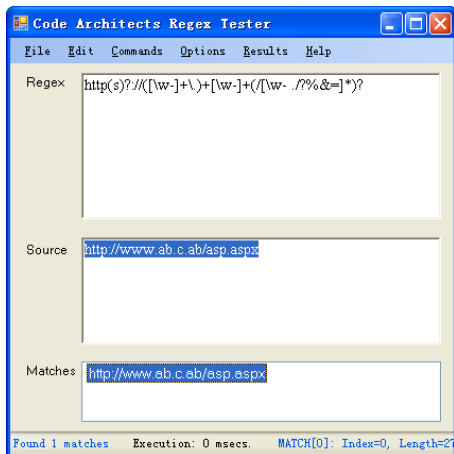


图 3.37 测试正则表达式 http(s)?://([\w-]+\.)+[\w-]+(/[\w- ./?%&=]*)?

3.4.4 FTP 地址验证

FTP 地址一般是以字符串“ftp://”开头的字符串。它和 HTTP 地址最大的区别是：FTP 地址以字符串“ftp”开头，而 HTTP 地址以字符串“http”开头。同样地，FTP 地址也可以被字符、/、?、&、%和=分割，如 ftp://www.abc.com、ftp://abc.cn 和 ftp://www.a.b.net/2007/aa.aspx 等。由验证 HTTP 地址的正则表达式可知，以下正则表达式能够验证 FTP 地址。

```
ftp://([\w-]+\.)+[\w-]+(/[\w- ./?%&=]*)? (77)
```

1. 示例（77）讲解

示例（77）的讲解如下。

- `[\w-]`能够匹配单词字符和连接符号-。
- `\.`匹配字符。
- `[\w-]+\.`能够匹配由单词字符和连接符号-组成的以字符串开头的、以字符.结尾的字符串。
- `([\w-]+\.)+`能够匹配 1 个或多个由单词字符和连接符号-组成的以字符串开头的、以字符.结尾的字符串。
- `/`匹配字符/。
- `[\w- ./?%&=]`能够匹配单词字符、-、（空格）、.、/、?、%、&和=；`[\w- ./?%&=]*`能够匹配空字符串，或者由单词字符、-、（空格）、.、/、?、%、&、=组成的长度至少为 1 的字符串。
- `([\w- ./?%&=]*)?`表示表达式`[\w- ./?%&=]*`匹配的字符串可以不出现或者只出现 1 次。

使用工具 Regex Tester 测试正则表达式 `ftp://([\w-]+\.)+[\w-]+(/[\w- ./?%&=]*)?`，结果如图 3.38 所示。

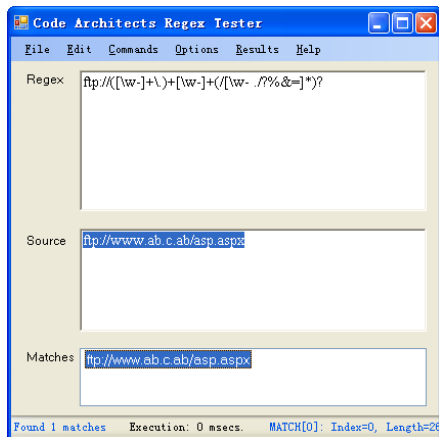


图 3.38 测试正则表达式 `ftp://([\w-]+\.)+[\w-]+(/[\w- ./?%&=]*)?`

第 4 章 常见数字和字符混合验证

本章的主要内容是介绍由数字和非数字字符组成的字符串的验证方法。这是一种混合验证，如数学表达式验证、日期和时间验证、编码规范验证、车牌号码验证等。

注意：本章中被验证的字符串可以由数字字符和英文单词字符或特殊字符组成。其中，特殊字符是指除英文单词字符和数字字符之外的字符，如/、\、|、,和:等。

4.1 5 种数学表达式验证

本节介绍与数学表达式相关的验证，如运算数验证、运算符验证、简单数学表达式验证和包含小括号的数学表达式验证等。

4.1.1 运算数验证

在此，定义运算数包括变量运算数和数值运算数。其中，变量运算数是由单词字符构成的字符串；数值运算数包括整数和实数，如 10、a、leftValue 和 0.1 等。以下正则表达式能够验证由单词字符构成的运算数。

<code>\w+</code>	(1)
<code>\w{1,}</code>	(2)

以下正则表达式能够验证数值运算数（包括整数和实数）。

<code>-?(0 ([1-9]\d*))(\.\d+)?</code>	(3)
<code>-?(0 ([1-9]\d{0,}))(\.\d{1,})?</code>	(4)

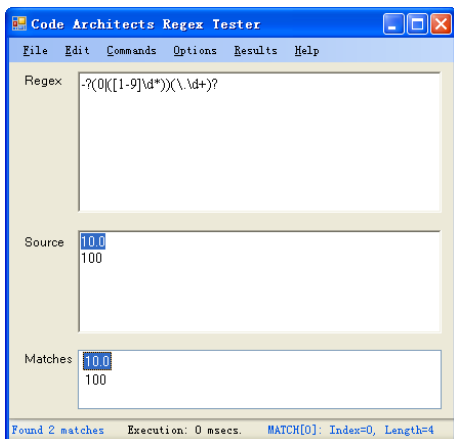
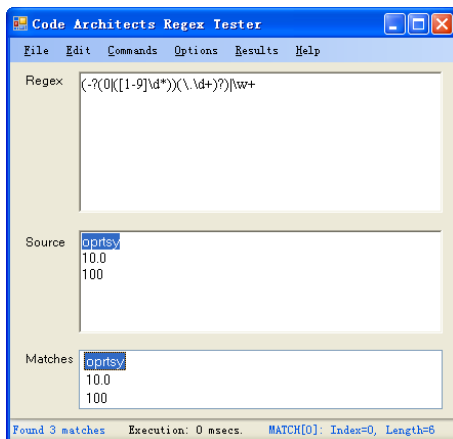
正则表达式解释如下。

- 正则表达式`-?(0|([1-9]\d*))(\.\d+)?`：`-?`可以匹配符号`-`，或者匹配空字符串；`0`匹配字符`0`；`[1-9]\d*`匹配第一个字符不为`0`的、由数字构成的字符串；`\.`匹配字符串.（小数点）；`\d+`匹配小数点之后的所有数字；`(\.\d+)?`可以匹配空字符串，或者匹配从小数点开始的所有字符。
- 正则表达式`-?(0|([1-9]\d{0,}))(\.\d{1,})?`：`-?`可以匹配符号`-`，或者匹配空字符串；`0`匹配字符`0`；`[1-9]\d{0,}`匹配第一个字符不为`0`的、由数字构成的字符串；`\.`匹配字符串.（小数点）；`\d{1,}`匹配小数点之后的所有数字；`(\.\d{1,})?`可以匹配空字符串，或者匹配从小数点开始的所有字符。

使用工具 Regex Tester 测试正则表达式`-?(0|([1-9]\d*))(\.\d+)?`，结果如图 4.1 所示。综上所述，以下正则表达式能够验证所有运算数。

<code>(-?(0 ([1-9]\d*))(\.\d+)?) \w+</code>	(5)
<code>(-?(0 ([1-9]\d{0,}))(\.\d{1,})?) \w{1,}</code>	(6)

使用工具 Regex Tester 测试正则表达式`(-?(0|([1-9]\d*))(\.\d+)?)|\w+`，结果如图 4.2 所示。

图 4.1 测试正则表达式 $-?(0|([1-9]d^*))(\.d+)?$ 图 4.2 测试正则表达式 $(-?(0|([1-9]d^*))(\.d+)?)\w^+$

4.1.2 运算符验证

在此，定义运算符包括+（加号）、-（减号）、*（乘号）和/（除号）。以下正则表达式能够验证运算符。

```
[ -+*/ ] (7)
```

4.1.3 简单数学表达式验证

在此，定义简单数学表达式是由两个运算数和一个运算符构成的表达式，并且表达式中不含任何括号。特别地，由于该表达式中不能包含任何括号，所以当某一个运算数为负数时，该运算数只能是表达式中的第一个运算数。下面是简单数学表达式的一些典型例子。

```
a+b
leftvalue*10
100-5
-12+rightvalue
```

以下正则表达式能够验证所有运算数。

```
(-?(0|([1-9]d*))(\.d+)?)\w+ (8)
```

以下正则表达式能够验证运算符。

```
[ -+*/ ] (9)
```

根据上述两个正则表达式可以得出，以下正则表达式能够验证简单数学表达式。

```
^((-?(0|([1-9]d*))(\.d+)?)\w+)[ -+*/ ]((0|([1-9]d*))(\.d+)?)\w+$ (10)
```

正则表达式解释如下。

- $-?(0|([1-9]d^*))(\.d+)?$ 可以验证所有实数和整数，即数值运算数。
- \w^+ 可以验证由单词字符构成的运算数，即变量运算数。
- $[-+*/]$ 匹配运算符。
- $(-?(0|([1-9]d^*))(\.d+)?)\w^+$ 可以匹配左运算数。
- $(0|([1-9]d^*))(\.d+)?\w^+$ 可以匹配右运算数。

使用工具 Regex Tester 测试正则表达式 $^((-?(0|([1-9]d^*))(\.d+)?)\w^+)[-+*/]((0|([1-9]d^*))(\.d+)?)\w^+$，结果如图 4.3 所示。$

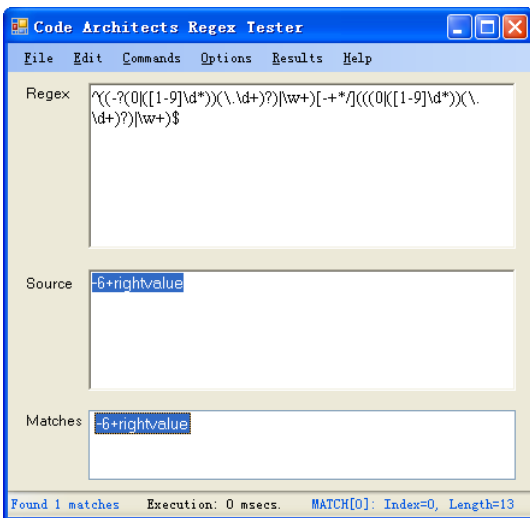


图 4.3 测试正则表达式 $^((-?(0|([1-9]\d*))(\.\d+)?)\w+)([-+*/](((0|([1-9]\d*))(\.\d+)?)\w+))\$$

4.1.4 只含运算数和运算符的数学表达式验证

在此，定义只含运算数和运算符的数学表达式是由多个运算数和多个运算符构成的表达式，并且表达式中不含任何括号。特别地，由于该表达式中不能包含任何括号，所以当某一个运算数为负数时，该运算数只能是表达式中的第一个运算数。下面是只含数字和运算符的数学表达式的一些典型例子。

```
a+b+c+d
leftvalue*10-100
100-5+v
-12+rightvalue+a+b-c*v
```

根据前面两个小节的内容可知，以下正则表达式能够验证所有运算数。

```
(-?(0|([1-9]\d*))(\.\d+)?)\w+ (11)
```

以下正则表达式能够验证运算符。

```
[-+*/] (12)
```

由上述两个正则表达式可知，以下正则表达式能够验证只含运算数和运算符的数学表达式。

```
^((-?(0|([1-9]\d*))(\.\d+)?)\w+)([-+*/](((0|([1-9]\d*))(\.\d+)?)\w+))+ (13)
```

正则表达式解释如下。

- $-?(0|([1-9]\d*))(\.\d+)?$ 可以验证所有实数和整数，即数值运算数。
- $\w+$ 可以验证由单词字符构成的运算数，即变量运算数。
- $[-+*/]$ 匹配运算符。
- $(-?(0|([1-9]\d*))(\.\d+)?)\w+$ 可以匹配左运算数。
- $(0|([1-9]\d*))(\.\d+)?)\w+$ 可以匹配右运算数。
- $([-+*/](((0|([1-9]\d*))(\.\d+)?)\w+))+$ 可以匹配多个（至少为1个）由“运算符+运算数”组成的字符串。

使用工具 **Regex Tester** 测试正则表达式 $^((-?(0|([1-9]\d*))(\.\d+)?)\w+)([-+*/](((0|([1-9]\d*))(\.\d+)?)\w+))+\$$ ，结果如图 4.4 所示。

图 4.4 测试正则表达式 $^((-?(0|([1-9]d*))(\.d+)?)\|w+)([+*/](((0|([1-9]d*))(\.d+)?)\|w+)))+\$$

4.1.5 包含小括号的数学表达式验证

如果被验证的数学表达式包含小括号，那么在验证该表达式时需要验证小括号是否匹配。否则，该表达式是一个不合法的数学表达式。验证该类型的数学表达式时需要使用递归匹配语法。以下正则表达式能够验证包含小括号的数学表达式。

```
(?<char>\(?)[^()]*(((?<bracket>\([^\()]*)+((?<-bracket>\)))[^()]*+)*(?(bracket)(?!))\k<char> (14)
```

以下正则表达式添加了注释，它和示例（14）等价，也能够验证包含小括号的数学表达式。

```
(?<char>\(?           # 匹配最外层的左括号，如果存在
[^()]*              # 匹配最外层左括号后面的、不是括号“()”的内容
(
  (
    (?<bracket>\(      # 如果匹配到左括号，则命名为 bracket，并压入堆栈
    [^()]*          # 匹配当前左括号后面的、不是括号“()”的内容
  )+
  (
    (?<-bracket>\)     # 如果匹配到右括号，则弹出命名为 bracket 的内容
    [^()]*          # 匹配右括号后面不是括号的内容
  )+
  *(?(bracket)(?!))  # 如果匹配到最外层的右括号前面，则检查堆栈内容
                        # 是否为空。如果不为空，则匹配失败
)\k<char>           # 匹配最外层的右括号，如果存在
```

使用工具 **Regex Tester** 测试正则表达式示例（14），结果如图 4.5 所示。

正则表达式 $(?<char>\(?)[^()]*(((?<bracket>\([^\()]*)+((?<-bracket>\)))[^()]*+)*(?(bracket)(?!))\k<char>$ 只能简单验证包含小括号的数学表达式。准确地说，它能够匹配小括号对称的表达式。因此，它还能够匹配表达式“(a=200)*(20-5)+1”，测试结果如图 4.6 所示。显然，该表达式不是一个合法的数学表达式。

为了更加精确地验证包含小括号的数学表达式，需要使用前面两个小节内容中验证运算数和运算符的正则表达式。根据前面两个小节内容可知，以下正则表达式能够验证所有运算数。

```
(-?(0|([1-9]d*))(\.d+)?)\|w+ (15)
```


- [89]匹配年字符串的第二位数字，它可以是 8 或者 9。
- \d{2}匹配年字符串的后面两位数字，它可以是任何数字。

2. 示例（22）讲解

示例（22）的讲解如下。

- ^和\$分别匹配字符串的开始位置和结束位置。
- 2 匹配年字符串左边的第一位数字。
- [0-4]匹配年字符串的第二位数字，它可以是 0、1、2、3 或 4。
- \d{2}匹配年字符串的后面两位数字，它可以是任何数字。

使用工具 Regex Tester 测试正则表达式`^(2500|(2[0-4]\d{2}))(1[89]\d{2}))$`，结果如图 4.8 所示。

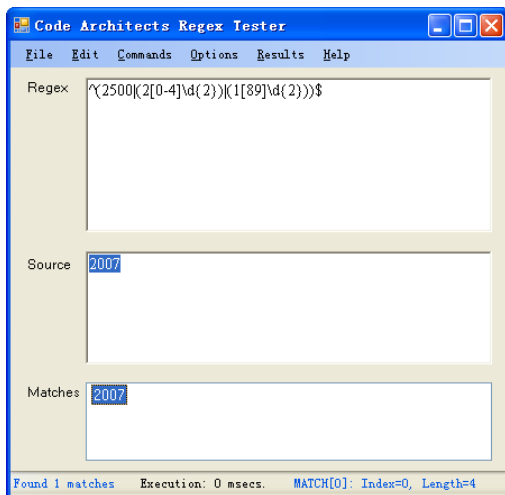


图 4.8 测试正则表达式`^(2500|(2[0-4]\d{2}))(1[89]\d{2}))$`

4.2.2 月验证

月的字符串为一个固定范围内的整数，该范围为 1~12。以下正则表达式能够验证月的字符串。

`^([1-9]|(1[0-2]))$` (25)

`^([1-9]|(1(0|1|2)))$` (26)

1. 示例（25）和（26）讲解

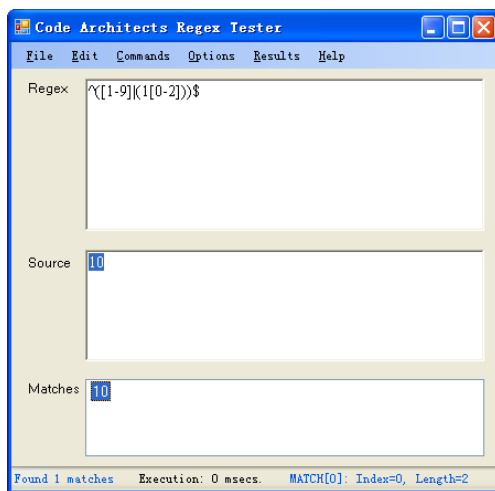
示例（25）的讲解如下。

- ^和\$分别匹配字符串的开始位置和结束位置。
- [1-9]匹配 1~9 中的任何一个月的字符串。
- 1[0-2]匹配 10、11 或 12 月的字符串。

示例（26）的讲解如下。

- ^和\$分别匹配字符串的开始位置和结束位置。
- [1-9]匹配 1~9 中的任何一个月的字符串。
- 1(0|1|2)匹配 10、11 或 12 月的字符串。

使用工具 Regex Tester 测试正则表达式`^([1-9]|(1[0-2]))$`，结果如图 4.9 所示。

图 4.9 测试正则表达式`^([1-9])([0-2])$`

4.2.3 日验证

日的字符串为一个固定范围内的整数，该范围为 1~31。以下正则表达式能够验证日的字符串。

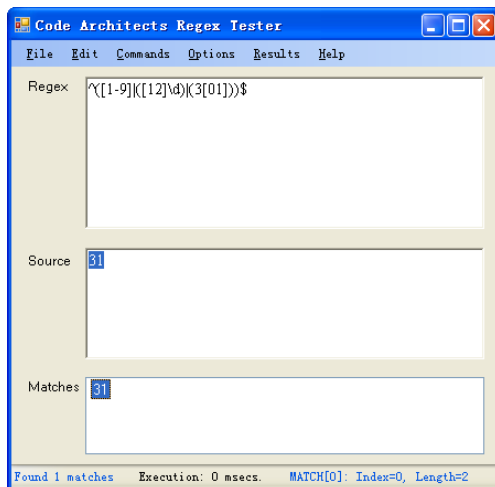
`^([1-9]|([12]\d)|(3[01]))$` (27)

`^([1-9]|((1|2)\d)|(3(0|1)))$` (28)

正则表达式解释如下。

- 正则表达式`^([1-9]|([12]\d)|(3[01]))$`：`^`和`$`分别匹配字符串的开始位置和结束位置；`[1-9]`匹配 1~9 中的任何一个日的字符串；`[12]\d`匹配 10~29 范围内的任何一个日的字符串；`3[01]`匹配 30 或者 31 日。
- 正则表达式`^([1-9]|((1|2)\d)|(3(0|1)))$`：`^`和`$`分别匹配字符串的开始位置和结束位置；`[1-9]`匹配 1~9 中的任何一个日的字符串；`(1|2)\d`匹配 10~29 范围内的任何一个日的字符串；`3(0|1)`匹配 30 或者 31 日。

使用工具 Regex Tester 测试正则表达式`^([1-9]|([12]\d)|(3[01]))$`，结果如图 4.10 所示。

图 4.10 测试正则表达式`^([1-9]|([12]\d)|(3[01]))$`

4.2.4 年月日格式的日期验证

年月日格式的日期一般由年、月、日的字符串，以及它们之间的连接符号（如-、/、.和空白字符等）组成。如果年、月、日字符串之间的连接符号为“-”，则以下正则表达式能够验证年月日格式的日期的字符串。

```
^\d{4}-([1-9]|(1[0-2]))-([1-9]|([12]\d)|(3[01]))$ (29)
```

```
^\d{4}-([1-9]|(1[0-2]))-([1-9]|((1|2)\d)|(3(0|1))))$ (30)
```

1. 示例（29）讲解

示例（29）的讲解如下。

- ^和\$分别匹配字符串的开始位置和结束位置。
- \d{4}匹配长度为4的年字符串。
- ([1-9]|(1[0-2]))匹配1~12范围内的月份。
- ([1-9]|((1|2)\d)|(3(0|1)))匹配日期。其中，[1-9]匹配1~9中的任何一个日的字符串；(1|2)\d匹配10~29范围内的任何一个日的字符串；3(0|1)匹配30或者31日。

使用工具 Regex Tester 测试正则表达式`^\d{4}-([1-9]|(1[0-2]))-([1-9]|([12]\d)|(3[01]))$`，结果如图4.11所示。

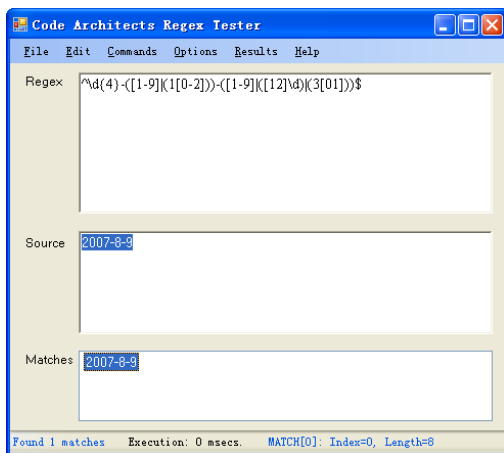


图 4.11 测试正则表达式`^\d{4}-([1-9]|(1[0-2]))-([1-9]|([12]\d)|(3[01]))$`

正则表达式`^\d{4}-([1-9]|(1[0-2]))-([1-9]|([12]\d)|(3[01]))$`只能匹配形如“2007-8-9”格式的日期。然而，如“2007-08-9”、“2007-8-09”、“2007/8/9”、“2007 8 9”和“2007.8.9”等格式的日期都不能被匹配，问题如下。

- 示例（30）中的年、月、日之间的连接符号固定为字符“-”；
- 当表示月、日的整数小于10时，示例（30）只能匹配一位整数，而不能匹配以0开头的两位数字字符串。

以下正则表达式能够验证年月日格式的日期的字符串，且年、月、日之间的连接符号可以为-、/、.或空白字符。

```
^\d{4}[-./ ]([1-9]|(1[0-2]))[-./ ]([1-9]|([12]\d)|(3[01]))$ (31)
```

上述正则表达式能够匹配形如“2007-8/9”格式的日期，这是不能接受的。为了解决这一问题，可以使用后向引用。以下正则表达式能够验证年月日格式的日期的字符串，年、月、日之间的连接符号可以为-、/、.或空白字符，且年、月之间的连接符号和月、日之间的连接符号相同。

```
^\d{4} (?<connectchar>[-/. ])([1-9] | (1[0-2])) \k<connectchar> ([1-9] | ([12]\d) | (3[01]))$ (32)
```

为了解决另外一个问题，即需要能够匹配字符串 8，又能够匹配字符串 08 的正则表达式。以下正则表达式能够验证 1~12 范围内的月的字符串。其中，月的字符串可以表示为 8、08 或 12。

```
(1[0-2]) | (0?\d) (33)
```

以下正则表达式能够验证 1~31 范围内的日的字符串。其中，日的字符串可以表示为 8、08 或 31。

```
([12]\d) | (3[01]) | (0?\d) (34)
```

综上所述，可以得到一个比较精确的验证年月日格式的日期的正则表达式，具体如下。

```
^\d{4} (?<connectchar>[-/. ])(1[0-2]) | (0?\d) \k<connectchar> (([12]\d) | (3[01]) | (0?\d))$ (35)
```

使用工具 Regex Tester 测试正则表达式`^\d{4} (?<connectchar>[-/.])((1[0-2])|(0?\d))\k<connectchar> (([12]\d)|(3[01])|(0?\d))$`，结果如图 4.12 所示。

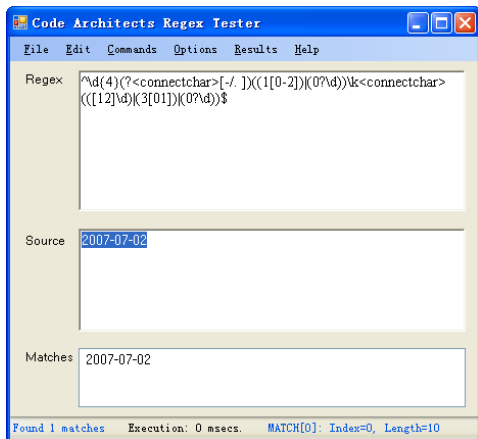


图 4.12 测试正则表达式`^\d{4} (?<connectchar>[-/.])((1[0-2])|(0?\d))\k<connectchar> (([12]\d)|(3[01])|(0?\d))$`

4.2.5 24 小时制时分秒格式的时间验证

本节讲解的字符串是 24 小时制时分秒格式的时间字符串，它的格式为“hh:mm:ss”。小时（hh）、分（mm）和秒（ss）的字符串的长度都为 2，且使用冒号（:）分割。另外，小时应在 0~23 范围之内，分和秒都应在 0~59 范围之内。以下正则表达式能够验证在 0~23 范围内的小时的字符串。

```
([0-1]\d) | (2[0-3]) (36)
```

```
([0-1][0-9]) | (2[0-3]) (37)
```

以下正则表达式能够验证在 0~59 范围内的分或者秒的字符串。

```
[0-5]\d (38)
```

```
[0-5][0-9] (39)
```

注意：正则表达式（36）、（37）、（38）和（39）验证的小时、分、秒的字符串的长度都为 2。如果要想使小时、分、秒的字符串的长度为 1（当小时、分、秒的值小于 10 时，它可以直接使用一位整数表示），则需要调整正则表达式。

以下正则表达式能够验证在 0~23 范围内的整数，小于 10 的整数使用一位数字表示。

```
(1?\d) | (2[0-3]) (40)
```

```
(1?[0-9]) | (2[0-3]) (41)
```


综上所述，以下正则表达式能够验证 24 小时制时分秒格式的时间字符串，且该字符串的格式为 hh:mm:ss。

```
(([0-1]\d)|(2[0-3]))(:[0-5]\d){2} (42)
```

```
(([0-1][0-9])|(2[0-3]))(:[0-5][-9]){2} (43)
```

正则表达式解释如下。

- ❑ 正则表达式`(([0-1]\d)|(2[0-3]))`：`[0-1]\d` 验证了以数字 0 或者 1 开头的小于 20 的整数字符串，且第一位可以为 0；`2[0-3]`可以匹配 20、21、22 和 23 这四个数字。因此，该正则表达式可以匹配 0~23 范围内的任意整数字符串。
- ❑ 正则表达式`(:[0-5]\d)`：`[0-5]`可以匹配 0~5 中的任意数字。该正则表达式可以匹配 0~59 范围内的任意整数字符串，且第一位可以为 0。
- ❑ 正则表达式`(:[0-5]\d){2}`：`: [0-5]\d` 可以匹配冒号+0~59 范围内的任意整数字符串；`(:[0-5]\d){2}`重复冒号+0~59 范围内的任意整数字符串两次，匹配时间中的分和秒部分。

使用工具 Regex Tester 测试正则表达式`(([0-1]\d)|(2[0-3]))(:[0-5]\d){2}`，结果如图 4.13 所示。

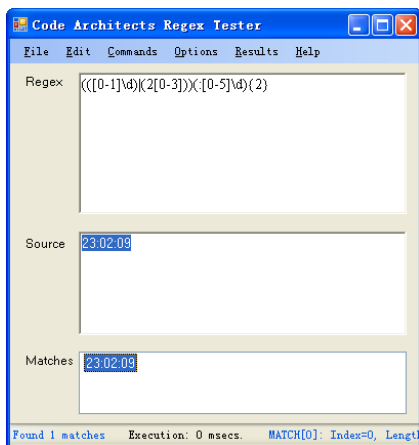


图 4.13 测试正则表达式`(([0-1]\d)|(2[0-3]))(:[0-5]\d){2}`

4.2.6 12 小时制时分秒格式的时间验证

本节讲解的字符串是 12 小时制时分秒格式的时间字符串，它的格式为“hh:mm:ss am”或者为“hh:mm:ss pm”。小时（hh）、分（mm）和秒（ss）的字符串的长度都为 2，且使用冒号（:）分割。另外，小时应在 0~12 范围之内，分和秒都应在 0~59 范围之内。以下正则表达式能够验证在 0~12 范围之内的小时的字符串。

```
(0\d)|(1[0-2]) (44)
```

```
(0[0-9])|(1[0-2]) (45)
```

以下正则表达式能够验证在 0~59 范围之内分的字符串。

```
[0-5]\d (46)
```

```
[0-5][0-9] (47)
```

注意：正则表达式（44）、（45）、（46）和（47）验证的小时、分、秒的字符串的长度都为 2。如果允许小时、分、秒、的字符串的长度为 1（当小时、分、秒的值小于 10 时，它可以直接使用一位整数表示），则需要调整正则表达式。

以下正则表达式能够验证 0~12 范围之内整数，小于 10 的整数使用一位数字表示。

```
(\d)|(1[0-2]) (48)
([0-9])|(1[0-2]) (49)
```

综上所述，以下正则表达式能够验证 12 小时制的时分秒格式的时间字符串。它的格式为“hh:mm:ss am”或者为“hh:mm:ss pm”。

```
((0\d)|(1[0-2]))(:[0-5]\d){2} (a|p)m (50)
```

```
((0[0-9])|(1[0-2]))(:[0-5]\d){2} (a|p)m (51)
```

正则表达式解释如下。

- ❑ 正则表达式 $(0\d)|(1[0-2])$ ： $0\d$ 验证了以数字 0 开头的小于 10 的整数字符串，且第一位可以为 0； $1[0-2]$ 可以匹配 10、11、22 这 3 个数字。因此，该正则表达式可以匹配 0~12 范围内的任意整数字符串。
- ❑ 正则表达式 $[0-5]\d$ ： $[0-5]$ 可以匹配 0~5 中的任意数字。该正则表达式可以匹配 0~59 范围内的任意整数字符串，且第一位可以为 0。
- ❑ 正则表达式 $(:[0-5]\d){2}$ ： $: [0-5]\d$ 可以匹配冒号+0~59 范围内的任意整数字符串； $(:[0-5]\d){2}$ 重复冒号+0~59 范围内的任意整数字符串两次，匹配时间中的分和秒。
- ❑ 正则表达式 $(a|p)m$ ：第一个字符匹配空白字符； $(a|p)m$ 可以匹配 am 或者 pm。

使用工具 Regex Tester 测试正则表达式 $((0\d)|(1[0-2]))(:[0-5]\d){2} (a|p)m$ ，结果如图 4.14 所示。

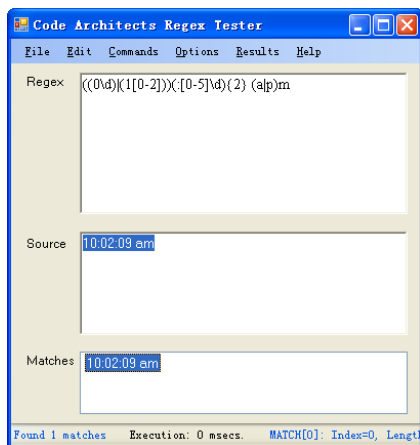


图 4.14 测试正则表达式 $((0\d)|(1[0-2]))(:[0-5]\d){2} (a|p)m$

4.2.7 带毫秒的时间验证

毫秒等于千分之一秒，它的数值应在 0~999 范围之内。以下正则表达式能够验证 0~999 范围内的毫秒的字符串。

```
([1-9]\d{1,2})|\d (52)
```

```
([1-9][0-9]{1,2})|[0-9] (53)
```

如果想要让 0~999 范围内的毫秒字符串的第一个字符可以为数字 0，则需要新的正则表达式来验证该字符串。以下正则表达式能够验证在 0~999 范围内的毫秒字符串的第一个字符为数字 0 的字符串。

```
\d\d?\d? (54)
```

本节讲解的字符串是 24 或 12 小时制时分秒毫秒格式的时间字符串。

1. 24 小时制

如果为 24 小时制，则它的格式为“hh:MM:ss:mmm”。小时(hh)、分(MM)和秒(ss)

的字符串的长度都为 2，且使用冒号（:）分割。另外，小时应在 0~23 范围之内，分和秒都应在 0~59 范围之内；毫秒应在 0~999 范围之内，且第一个字符可以为数字 0。以下正则表达式能够验证 24 小时制时分秒毫秒格式的时间字符串，且该字符串的格式为 hh:MM:ss:mmm。

```
(([0-1]\d)|(2[0-3]))(:[0-5]\d){2}:\d\d?\d? (55)
```

```
(([0-1][0-9])|(2[0-3]))(:[0-5][-9]){2}:\d\d?\d? (56)
```

正则表达式解释如下。

- 正则表达式`(([0-1]\d)|(2[0-3]))`：`[0-1]\d` 验证了以数字 0 或者 1 开头的小于 20 的整数字符串，且第一位可以为 0；`2[0-3]`可以匹配 20、21、22 和 23 这四个数字。因此，该正则表达式可以匹配 0~23 内的任意整数字符串。
- 正则表达式`[0-5]\d`：`[0-5]`可以匹配 0~5 中的任意数字。该正则表达式可以匹配 0~59 范围内的任意整数字符串，且第一位可以为 0。
- 正则表达式`(:[0-5]\d){2}`：`:[0-5]\d` 可以匹配冒号+0~59 范围内的任意整数字符串；`(:[0-5]\d){2}`重复冒号+0~59 范围内的任意整数字符串两次，匹配时间中的分和秒。
- 正则表达式`:\d\d?\d?`：可以匹配 000~999 范围内的任意一个数字字符串。

使用工具 Regex Tester 测试正则表达式`(([0-1]\d)|(2[0-3]))(:[0-5]\d){2}:\d\d?\d?`，结果如图 4.15 所示。

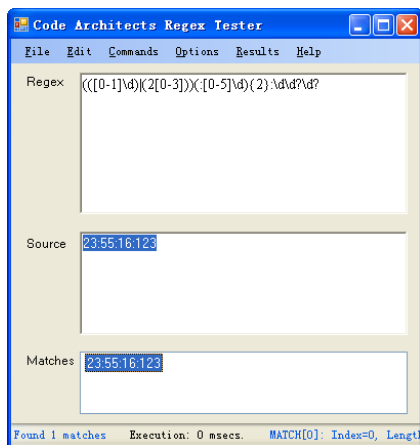


图 4.15 测试正则表达式`(([0-1]\d)|(2[0-3]))(:[0-5]\d){2}:\d\d?\d?`

2. 12 小时制

如果为 12 小时制，则它的格式为“hh:MM:ss:mmm am”或者“hh:MM:ss:mmm pm”。小时（hh）、分（MM）和秒（ss）的字符串的长度都为 2，且使用冒号（:）分割。另外，小时应在 0~12 范围之内，分和秒都应在 0~59 范围之内，毫秒应在 0~999 范围之内。以下正则表达式能够验证 12 小时制时分秒毫秒格式的时间字符串，它的格式为“hh:MM:ss:mmm am”或者“hh:MM:ss:mmm pm”。

```
((0\d)|(1[0-2]))(:[0-5]\d){2}:\d\d?\d? (a|p)m (57)
```

```
((0[0-9])|(1[0-2]))(:[0-5]\d){2}:\d\d?\d? (a|p)m (58)
```

正则表达式解释如下。

- 正则表达式`(0\d)|(1[0-2])`：`0\d` 验证了以数字 0 开头的小于 10 的整数字符串，且第一位可以为 0；`1[0-2]`可以匹配 10、11、12 这 3 个数字。因此，该正则表达式可以匹配 0~12 内的任意整数字符串。

- ❑ 正则表达式`[0-5]d`：`[0-5]`可以匹配 0~5 中的任意数字。该正则表达式可以匹配 0~59 范围内的任意整数字符串，且第一位可以为 0。
- ❑ 正则表达式`(:[0-5]d){2}`：`:[0-5]d` 可以匹配冒号+0~59 范围内的任意整数的字符串；`(:[0-5]d){2}`重复冒号+0~59 范围内的任意整数字符串两次，匹配时间中的分和秒。
- ❑ 正则表达式`\d\d?\d?`：可以匹配 000~999 范围内的任意一个数字字符串。
- ❑ 正则表达式 `(a)p)m`：第一个字符匹配空白字符；`(a)p)m` 可以匹配 am 或者 pm。

使用工具 Regex Tester 测试正则表达式`((0\d)(1[0-2]))(:[0-5]d){2}:\d\d?\d? (a)p)m`，结果如图 4.16 所示。

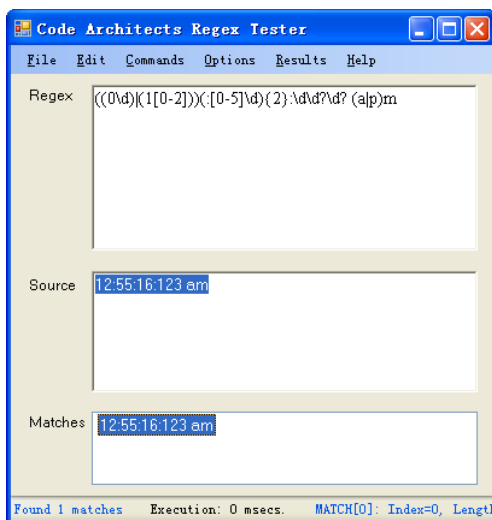


图 4.16 测试正则表达式`((0\d)(1[0-2]))(:[0-5]d){2}:\d\d?\d? (a)p)m`

4.2.8 长格式的日期和时间验证

本节讲解的字符串是长格式的日期（包含年月日）和时间（24 小时制）字符串，它的格式为“yyyy-MM-dd hh:mm:ss”。年（yyyy）、月（MM）、日（dd）的字符串的长度分别为 4、2、2，它们之间的连接符号可以是-、/或（空格）等。小时（hh）、分（mm）和秒（ss）的字符串的长度都为 2，且使用冒号（:）分割。另外，小时应在 0~23 范围之内，分和秒都应在 0~59 范围之内。

前面小节中已经介绍了年月日格式的日期验证方法，以下正则表达式能够验证年月日格式的日期字符串。

```
^\\d{4}(?<connectchar>[-/\\s])((1[0-2])|(0?\\d))\\k<connectchar>(((12)\\d)|(3[01])|(0?\\d))$ (59)
```

在 4.2.5 节中已经介绍了 24 小时制的时间字符串的时间验证方法，以下正则表达式能够验证 24 小时制的时间字符串。

```
(([0-1]\\d)|(2[0-3]))(:[0-5]\\d){2} (60)
```

```
(([0-1][0-9])|(2[0-3]))(:[0-5][0-9]){2} (61)
```

综上所述，以下正则表达式能够验证长格式的日期和时间的字符串。

```
^\\d{4}(?<connectchar>[-/\\s])((1[0-2])|(0?\\d))\\k<connectchar>(((12)\\d)|(3[01])|(0?\\d))((([0-1]\\d)|(2[0-3]))(:[0-5]\\d){2})$ (62)
```

使用工具 Regex Tester 测试正则表达式示例（62），结果如图 4.17 所示。

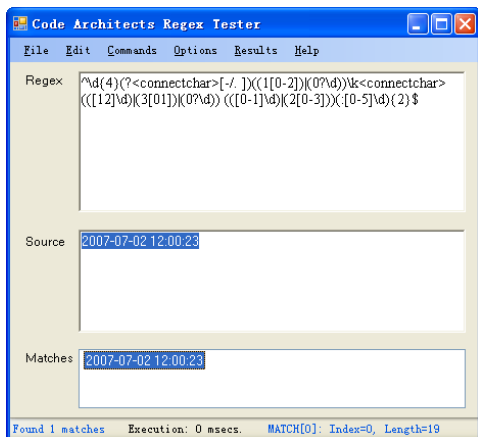


图 4.17 测试正则表达式示例 (62)

4.3 4 种编码规范验证

本节介绍编码规范中部分语句的验证，如类名称的验证、函数名称的验证、声明变量表达式的验证和声明函数表达式的验证等。

4.3.1 类名称验证

类名称是一个字符串，且满足以下两个条件。

- 由单词字符组成。
- 开头字符只能为下画线或者字母。

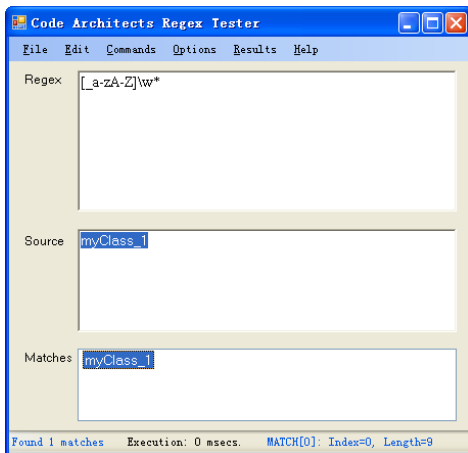
以下正则表达式能够验证类名称字符串。

```
[_a-zA-Z]\w* (63)
```

正则表达式解释如下。

- 字符类 `[_a-zA-Z]` 能够匹配一个下画线或者字母，它匹配类名称的第一个字符。
- `\w*` 匹配类名称除第一个字符之外的所有字符。

使用工具 Regex Tester 测试正则表达式 `[_a-zA-Z]\w*`，结果如图 4.18 所示。

图 4.18 测试正则表达式 `[_a-zA-Z]\w*`

4.3.2 声明变量表达式验证

在此，定义声明变量表达式格式为“变量类型+变量名称”。其中，变量类型字符串和变量名称都满足以下两个条件。

- 由单词字符组成。
- 开头字符只能为下画线或者字母。

以下正则表达式能够验证变量类型或变量名称的字符串。

```
[_a-zA-Z]\w* (64)
```

综合上述正则表达式可知，以下正则表达式能够验证声明变量表达式的字符串。

```
([_a-zA-Z]\w*\s+)+[_a-zA-Z]\w*; (65)
```

1. 示例（65）讲解

示例（65）的讲解如下。

- 字符类[_a-zA-Z]能够匹配一个下画线或者字母，它匹配变量类型字符串或变量名称的第一个字符。
- \w*匹配变量类型字符串或变量名称除第一个字符之外的所有字符。
- ([_a-zA-Z]\w*\s+)匹配一个形如“变量类型+空白字符”的字符串。
- ([_a-zA-Z]\w*\s+)+匹配变量前面的所有修饰符和变量类型组成的字符串。
- [_a-zA-Z]\w*;匹配变量名称和字符；。

使用工具 Regex Tester 测试正则表达式([_a-zA-Z]\w*\s+)+[_a-zA-Z]\w*;，结果如图 4.19 所示。

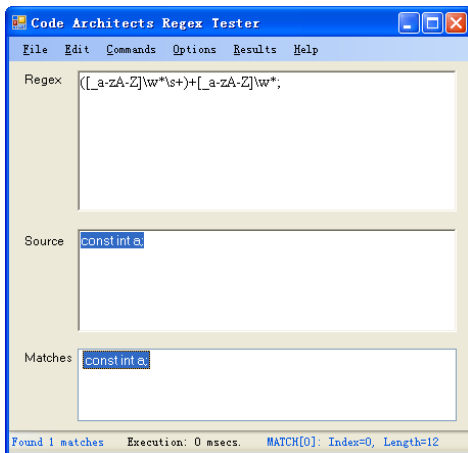


图 4.19 测试正则表达式([_a-zA-Z]\w*\s+)+[_a-zA-Z]\w*;

4.3.3 函数名称验证

函数名称是一个字符串，且满足以下两个条件。

- 由单词字符组成。
- 开头字符只能为下画线或者字母。

以下正则表达式能够验证函数名称字符串。

```
[_a-zA-Z]\w* (66)
```

正则表达式解释如下。

- 字符类[_a-zA-Z]能够匹配一个下画线或者字母，它匹配函数名称的第一个字符。

□ `\w*`匹配函数名称除第一个字符之外的所有字符。

使用工具 Regex Tester 测试正则表达式`[_a-zA-Z]\w*`，结果如图 4.20 所示。

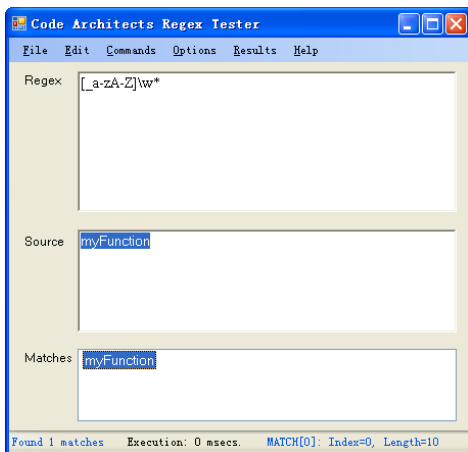


图 4.20 测试正则表达式`[_a-zA-Z]\w*`

4.3.4 声明函数表达式验证

声明一个函数表达式比声明一个变量表达式要复杂得多，它可以分为以下 6 种情况。

- 返回类型+函数名称()。
- 返回类型+函数名称（一个参数）。
- 返回类型+函数名称（多个参数）。
- 修饰符+返回类型+函数名称()。
- 修饰符+返回类型+函数名称（一个参数）。
- 修饰符+返回类型+函数名称（多个参数）。

由于函数的返回类型、修饰符与变量的变量类型、修饰符相似，因此，以下正则表达式可以验证函数前面的所有修饰符和返回类型组成的字符串。

```
([_a-zA-Z]\w*\s+)
```

 (67)

根据前面小节内容可知，以下正则表达式可以验证函数名称。

```
([_a-zA-Z]\w*)
```

 (68)

1. 声明无参数的函数表达式验证

如果函数无参数，那么该函数的名称之后接一对小括号。以下正则表达式可以验证声明无参数的函数表达式。

```
(([_a-zA-Z]\w*\s+)([_a-zA-Z]\w*\s*\(\s*\))
```

 (69)

2. 示例（69）讲解

示例（69）的讲解如下。

- 字符类`[_a-zA-Z]`能够匹配一个下画线或者字母，它匹配函数的修饰符、返回类型、名称的第一个字符。
- `([_a-zA-Z]\w*\s+)`匹配一个形如“返回类型+空白字符”或“修饰符+空白字符”的字符串。
- `([_a-zA-Z]\w*\s+)([_a-zA-Z]\w*\s*\(\s*\))`匹配函数前面的所有修饰符和返回类型组成的字符串。

- `[_a-zA-Z]\w*` 匹配函数的名称。
- `\s*(\s*)` 匹配小括号，以及字符(前后的空白字符)。

使用工具 Regex Tester 测试正则表达式 `([_a-zA-Z]\w*\s+)+[_a-zA-Z]\w*\s*(\s*)`，结果如图 4.21 所示。

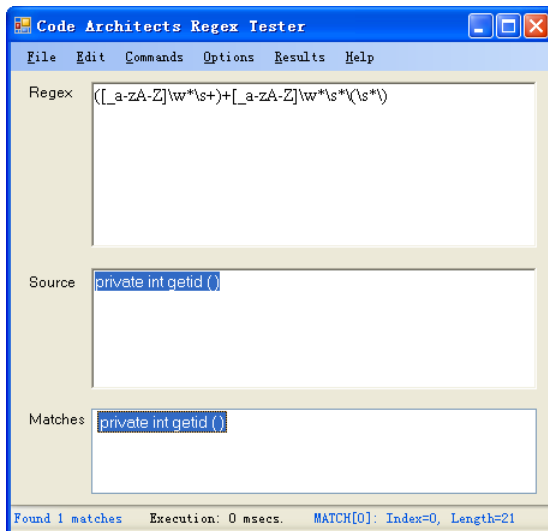


图 4.21 测试正则表达式 `([_a-zA-Z]\w*\s+)+[_a-zA-Z]\w*\s*(\s*)`

3. 声明含一个参数的函数表达式验证

如果函数包含一个参数，那么该函数的名称之后接一对小括号，且小括号中包含一个参数表达式。声明一个参数的表达式和声明一个变量的表达式非常相似。以下正则表达式能够验证声明一个参数的表达式。

```
([_a-zA-Z]\w*\s+)+[_a-zA-Z]\w* (70)
```

综合上述表达式可知，以下正则表达式可以验证声明含一个参数的函数表达式。

```
([_a-zA-Z]\w*\s+)+[_a-zA-Z]\w*\s*(\s*([_a-zA-Z]\w*\s+)+[_a-zA-Z]\w*\s*(\s*)) (71)
```

4. 示例（71）讲解

示例（71）的讲解如下。

- 字符类 `[_a-zA-Z]` 能够匹配一个下画线或者字母。
- `([_a-zA-Z]\w*\s+)` 匹配一个形如“返回类型+空白字符”、“修饰符+空白字符”或“变量类型+空白字符”的字符串。
- 第一个 `([_a-zA-Z]\w*\s+)+` 匹配函数前面的所有修饰符和返回类型组成的字符串。
- 第一个 `[_a-zA-Z]\w*` 匹配函数的名称。
- `\s*(\s*)` 匹配小括号左边部分，以及字符(前后的空白字符)。
- `([_a-zA-Z]\w*\s+)+[_a-zA-Z]\w*` 匹配声明函数参数的表达式。
- `\s*(\s*)` 匹配小括号右边部分，以及字符)前的空白字符。

使用工具 Regex Tester 测试正则表达式示例（71），结果如图 4.22 所示。

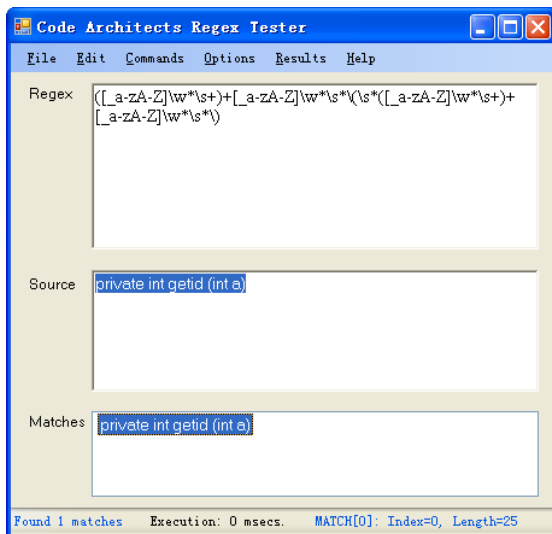


图 4.22 测试正则表达式示例 (71)

5. 声明含多个参数的函数表达式验证

如果函数含多个参数,那么该函数的名称之后接一对小括号,且小括号中包含多个参数表达式,各个参数表达式被字符,(逗号)分割。以下正则表达式能够验证声明一个参数的表达式。

```
([_a-zA-Z]\w*\s+)+[_a-zA-Z]\w* (72)
```

以下正则表达式能够验证声明多个参数的表达式。

```
([_a-zA-Z]\w*\s+)+[_a-zA-Z]\w*(\s*,\s*([_a-zA-Z]\w*\s+)+[_a-zA-Z]\w*\s*)* (73)
```

以下正则表达式可以验证声明多个参数的函数表达式。

```
([_a-zA-Z]\w*\s+)+[_a-zA-Z]\w*\s*(\s*([_a-zA-Z]\w*\s+)+[_a-zA-Z]\w*(\s*,\s*([_a-zA-Z]\w*\s+)+[_a-zA-Z]\w*\s*)*\s*) (74)
```

6. 示例 (74) 讲解

示例 (74) 的讲解如下。

- 字符类[_a-zA-Z]能够匹配一个下画线或者字母。
- ([_a-zA-Z]\w*\s+)匹配一个形如“返回类型+空白字符”、“修饰符+空白字符”或“变量类型+空白字符”的字符串。
- 第一个([_a-zA-Z]\w*\s+)+匹配函数前面的所有修饰符和返回类型组成的字符串。
- 第一个[_a-zA-Z]\w*匹配函数的名称。
- \s*(\s*匹配小括号左边部分,以及字符(前后的空白字符。
- ([_a-zA-Z]\w*\s+)+[_a-zA-Z]\w*(\s*,\s*([_a-zA-Z]\w*\s+)+[_a-zA-Z]\w*\s*)*匹配函数的参数列表表达式。其中,第一个([_a-zA-Z]\w*\s+)+[_a-zA-Z]\w*匹配第一个参数表达式;\s*,\s*([_a-zA-Z]\w*\s+)+[_a-zA-Z]\w*\s*匹配一个形如“字符逗号+参数表达式”的字符串;\s*,\s*([_a-zA-Z]\w*\s+)+[_a-zA-Z]\w*\s*)*匹配由0个、1个或多个形如“字符逗号+参数表达式”组成的字符串,即匹配第一个参数之后的所有参数字符串。
- \s*)匹配小括号右边部分,以及字符)前的空白字符。

使用工具 Regex Tester 测试正则表达式示例 (74),结果如图 4.23 所示。

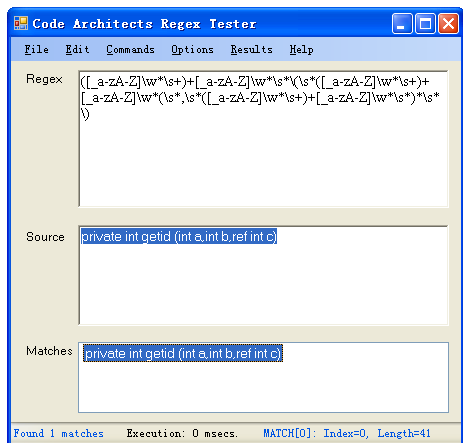


图 4.23 测试正则表达式示例（74）

7. 验证声明函数表达式总结

综合上述三种情况的验证，以下正则表达式能够验证声明函数的表达式。

`([_a-zA-Z]\w*\s+)+[_a-zA-Z]\w*\s*\(\s*(([_a-zA-Z]\w*\s+)+[_a-zA-Z]\w*\s*(\s*[_a-zA-Z]\w*\s*\s*(\s*[_a-zA-Z]\w*\s+)+[_a-zA-Z]\w*\s*)*\s*\)` (75)

使用工具 Regex Tester 测试正则表达式示例（75），结果如图 4.24 所示。

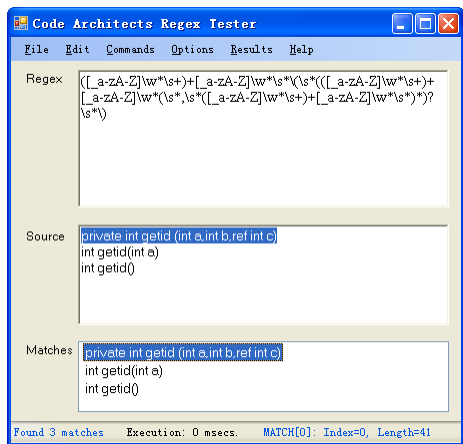


图 4.24 测试正则表达式示例（75）

4.4 3 种车牌号码验证

本节介绍车牌号码的验证方法，主要包括通用车牌号码、武警车牌号码和军用车牌号码的验证方法。

注意：如果车牌号码中包括直辖市或省的简称，那么在下面介绍的验证车牌号码的正则表达式中，不对直辖市或省的简称字符进行验证，只验证直辖市或省的简称之后的字符串。

4.4.1 通用车牌号码验证

目前，国内通用车牌号码主要有以下 4 种组成方式。

- ❑ 直辖市或省的简称+1 个大写英文字母+5 个数字。
- ❑ 直辖市或省的简称+2 个大写英文字母+4 个数字。
- ❑ 直辖市或省的简称+3 个大写英文字母+3 个数字。
- ❑ 直辖市或省的简称+1 个大写英文字母+1 个数字+1 个大写英文字母+3 个数字。

综上所述，通用车牌号码（不考虑直辖市或省的简称，下同）可以存在 4 种情况。以下正则表达式能够验证 1 个大写英文字母+5 个数字类型的车牌号码。

```
[A-Z]\d{5} (76)
```

```
[A-Z][0-9]{5} (77)
```

以下正则表达式能够验证 2 个大写英文字母+4 个数字类型的车牌号码。

```
[A-Z]{2}\d{4} (78)
```

```
[A-Z]{2}[0-9]{4} (79)
```

以下正则表达式能够验证 3 个大写英文字母+3 个数字类型的车牌号码。

```
[A-Z]{3}\d{3} (80)
```

```
[A-Z]{3}[0-9]{3} (81)
```

以下正则表达式能够验证 1 个大写英文字母+1 个数字+1 个大写英文字母+3 个数字的车牌号码。

```
[A-Z]\d[A-Z]\d{3} (82)
```

```
[A-Z][0-9][A-Z][0-9]{3} (83)
```

综上所述，以下正则表达式能够验证目前国内通用的车牌号码。

```
(([A-Z]\d{5})|([A-Z]{2}\d{4})|([A-Z]{3}\d{3})|([A-Z]\d[A-Z]\d{3})) (84)
```

正则表达式解释如下。

- ❑ 正则表达式`[A-Z]\d{5}`：`[A-Z]`能够匹配 A~Z 中的任何一个字母；`\d{5}`能够匹配长度为 5 的数字字符串。该表达式能够验证 1 个大写英文字母+5 个数字类型的车牌号码。
- ❑ 正则表达式`[A-Z]{2}\d{4}`：`[A-Z]{2}`能够匹配长度为 2、由大写字母组成的字符串；`\d{4}`能够匹配长度为 4 的数字字符串。该表达式能够验证 2 个大写英文字母+4 个数字类型的车牌号码。
- ❑ 正则表达式`[A-Z]{3}\d{3}`：`[A-Z]{3}`能够匹配长度为 3、由大写字母组成的字符串；`\d{3}`能够匹配长度为 3 的数字字符串。该表达式能够验证 3 个大写英文字母+3 个数字类型的车牌号码。

使用工具 Regex Tester 测试正则表达式示例（84），结果如图 4.25 所示。

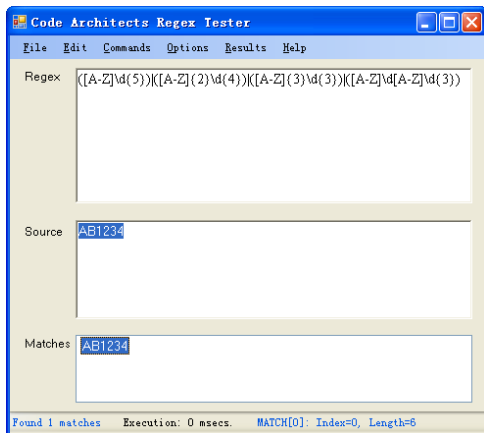


图 4.25 测试正则表达式示例（84）

除了上述介绍的验证国内通用车牌号码的方法之外，还有一个更加简单的验证方法。综合国内通用车牌号码的 4 种组合情况，可以得出其组成如下。

直辖市或省的简称+1 个大写英文字母+1 个大写英文字母或 1 个数字+1 个大写英文字母或 1 个数字+3 个数字。

因此，车牌号码的第 2 位是大写英文字母、第 3 位是大写英文字母或数字、第 4 位是大写英文字母或数字、第 5、6、7 位均为数字。以下正则表达式能够验证目前国内通用车牌号码。

```
[A-Z][A-Z0-9]{2}\d{3} (85)
```

正则表达式解释如下。

- 正则表达式 `[A-Z][A-Z0-9]{2}\d{3}`：`[A-Z]` 能够匹配 `A~Z` 中的任何一个字母；`[A-Z0-9]{2}` 能够匹配长度为 2 的、由大写字母或数字组成的字符串；`\d{3}` 能够匹配长度为 3 的数字字符串。该表达式能够验证 1 个大写英文字母+5 个数字类型的车牌号码。

使用工具 Regex Tester 测试正则表达式 `[A-Z][A-Z0-9]{2}\d{3}`，结果如图 4.26 所示。

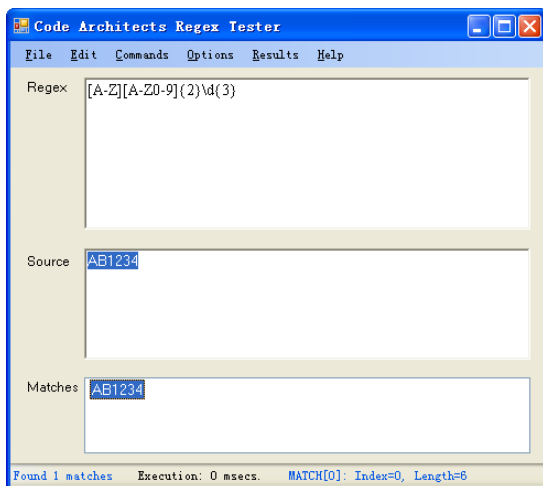


图 4.26 测试正则表达式 `[A-Z][A-Z0-9]{2}\d{3}`

4.4.2 武警车牌号码验证

目前国内武警车牌号码的组成格式为“WJ+两位省代码+连接符号-+5 位数字”。其中，WJ 表示“武警”；两位省代码为各个省份的代码，目前是 01~32。

在验证武警车牌号码之前，首先需要验证 01~32 范围内的两位省份代码。以下正则表达式能够验证 01~32 范围内的两位省份代码。

```
([0-2]\d)|(3[0-2]) (86)
```

```
([0-2][0-9])|(3[0-2]) (87)
```

综上所述，以下正则表达式能够验证目前国内武警车牌号码。

```
WJ(( [0-2]\d)|(3[0-2]))-\d{5} (88)
```

正则表达式解释如下。

- 正则表达式 `([0-2]\d)|(3[0-2])`：`[0-2]\d` 能够验证以 0、1 或 2 开头的、长度为 2 的数字字符串；`3[0-2]` 匹配 30、31 或 32。该表达式能够验证 01~32 范围内的两位省份代码。
- 正则表达式 `WJ(([0-2]\d)|(3[0-2]))-\d{5}`：WJ 匹配武警车牌号码的前两个字母；- 匹配连接符号-；`\d{5}` 能够匹配长度为 5 的数字字符串。

使用工具 Regex Tester 测试正则表达式 `WJ([0-2]\d)(3[0-2])-\d{5}`，结果如图 4.27 所示。



图 4.27 测试正则表达式 `WJ([0-2]\d)(3[0-2])-\d{5}`

第 5 章 常见的 HTML 元素验证和处理

本章主要介绍与 HTML 元素相关的验证。在介绍验证 HTML 的具体元素之前，首先介绍了 6 种 HTML 元素验证的基础；随后，介绍了部分 HTML 具体元素的验证，如
元素的验证、<hr>元素的验证、<a>元素的验证和<input>元素的验证等；最后，介绍了处理 HTML 元素的方法。

注意：为了方便书写正则表达式，特设定本章中所涉及的 HTML 元素包含的英文字母均为小写英文字母。

5.1 6 种 HTML 元素验证的基础

本节介绍了 HTML 元素的验证基础，如 HTML 标记验证、非封闭 HTML 标记验证、封闭 HTML 标记验证、属性赋值表达式验证、注释表达式验证和脚本代码验证等。

5.1.1 HTML 标记验证

HTML 标记一般是指被一对尖括号包围的、指定名称的字符串，如<a>、<hr>和<table>等。HTML 标记的名称中不能出现尖括号字符（包括<、>），且长度至少为 1。以下正则表达式能够简单验证 HTML 标记。

```
<.+> (1)
```

正则表达式<.+>能够简单验证 HTML 标记。该标记被尖括号包围，它的名称长度至少为 1。然而，若标记的名称中带有尖括号字符<或者>，则正则表达式<.+>不能正常工作。以下正则表达式能够简单验证 HTML 标记，且标记名称中不包括尖括号字符>。

```
<[^>]+> (2)
```

正则表达式解释如下。

- 正则表达式<.+>：.+表示除换行符号之外的字符至少出现 1 次；<匹配左边尖括号<；>匹配右边尖括号>。
- 正则表达式<[^>]+>：[^>]+表示除字符>之外的字符至少出现 1 次；<匹配左边尖括号<；>匹配右边尖括号>。

使用工具 **Regex Tester** 分别测试正则表达式<.+>和<[^>]+>，结果分别如图 5.1 和图 5.2 所示。

考虑 HTML 标记本身的特殊性，它的名称一般以英文字母开头，因此，以下正则表达式能够精确验证 HTML 标记，标记的名称必须以字母开头，且不能包括尖括号字符>。

```
<[a-z][^>]*> (3)
```

正则表达式解释如下。

- 正则表达式<[a-z][^>]*>：[a-z]匹配标记名称的第一个字符；[^>]+表示除字符>之外的字符至少出现 1 次；<匹配左边尖括号<；>匹配右边尖括号>。

使用工具 **Regex Tester** 测试正则表达式<[a-z][^>]*>，结果如图 5.3 所示。

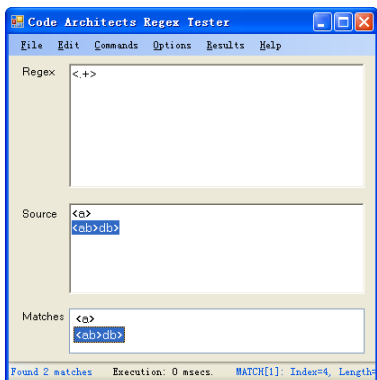


图 5.1 测试正则表达式<.+>

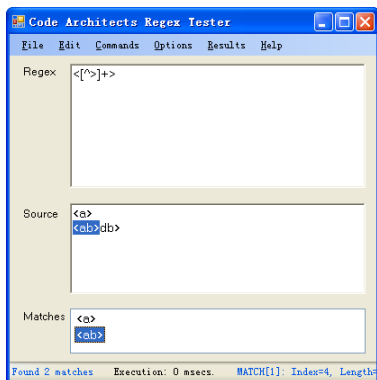


图 5.2 测试正则表达式<[<^>]+>

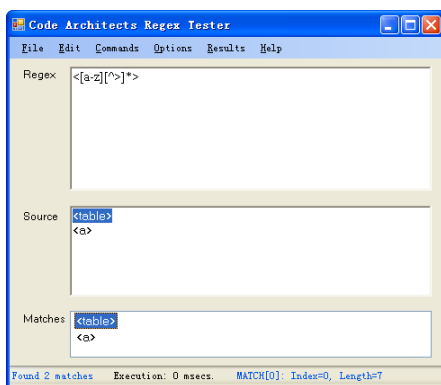


图 5.3 测试正则表达式<[a-z][^>]*>

5.1.2 非封闭 HTML 标记验证

一些 HTML 标记不要求封闭，即可以使用简单的/>来结束标记，如
等。另外，非封闭 HTML 标记又称为自封闭 HTML 标记。该类型的标记验证比较简单，只要在正则表达式<[a-z][^>]*>的第二个尖括号之前插入字符/即可。以下正则表达式能够精确验证自封闭的 HTML 标记，标记的名称必须以字母开头，且不能包括尖括号字符>。

```
<[a-z][^>]*>/>
```

(4)

使用工具 Regex Tester 测试正则表达式<[a-z][^>]*>/>，结果如图 5.4 所示。

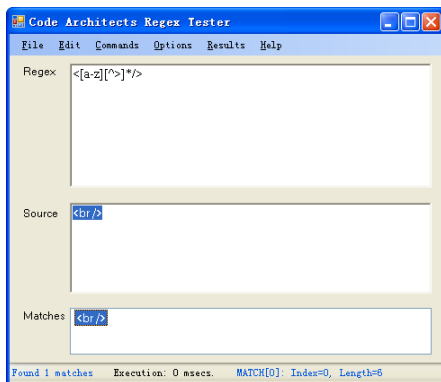


图 5.4 测试正则表达式<[a-z][^>]*>/>

5.1.3 封闭 HTML 标记验证

除了非封闭的 HTML 标记之外，还有一些 HTML 标记要求必须是封闭的，即开始标记和结束标记必须成对出现，如<table></table>、<td></td>等。

注意：本节介绍封闭 HTML 标记验证方法时，不考虑该标记中的具体内容。

5.1.1 节中已经介绍了验证 HTML 标记的正则表达式，具体如下。

```
<[a-z][^>]*> (5)
```

不妨设置被验证的封闭 HTML 标记为<table></table>，那么上述正则表达式只能验证开始标记部分，即<table>。以下正则表达式能够精确验证封闭 HTML 标记的结尾部分，标记的名称必须以字母开头，且不能包括尖括号字符>。

```
</[a-z][^>]*> (6)
```

综合上述两个正则表达式可知，以下正则表达式能够简单验证封闭 HTML 标记。

```
<[a-z][^>]*></[a-z][^>]*> (7)
```

上述正则表达式只能简单地验证封闭 HTML 标记。其实，它能够匹配形如“<table></td>”的 HTML 标记，即开始标记和结尾标记的名称不相同。这是在 HTML 标记验证中不允许的。为了解决这一问题，在此可以使用分组和后向引用的方法。以下正则表达式能够精确验证封闭 HTML 标记。

```
<(?(<name>[a-z][^>]*)></\k<name>> (8)
```

```
<([a-z][^>]*)></\1> (9)
```

正则表达式解释如下。

- ❑ 正则表达式<(?(<name>[a-z][^>]*)></\k<name>>：[a-z]匹配 a~z 的任何一个字符；[^>]*匹配由除字符>之外的字符组成的、最小长度为 0 的字符串；(?(<name>[a-z][^>]*)>)分组匹配标记的名称，并将该分组命名为“name”；</\k<name>>匹配标记的结束部分，并使用了名称为“name”的后向引用。
- ❑ 正则表达式<([a-z][^>]*)></\1>：[a-z]匹配 a~z 的任何一个字符；[^>]*匹配由除字符>之外的字符组成的、最小长度为 0 的字符串；([a-z][^>]*)分组匹配标记的名称，并将该分组命名为“\1”；</\1>匹配标记的结束部分，并使用了名称为“\1”的后向引用。

使用工具 Regex Tester 分别测试正则表达式示例（8）和示例（9），结果分别如图 5.5 和图 5.6 所示。

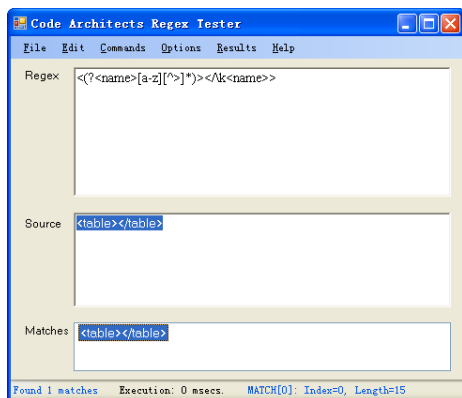


图 5.5 测试正则表达式示例（8）

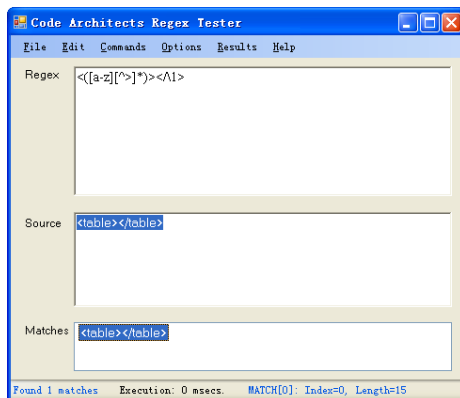


图 5.6 测试正则表达式示例（9）

5.1.4 属性赋值表达式验证

HTML 中的属性赋值表达式的格式可以分为以下几种情况。

- “属性名称=“值””
- “属性名称='值'”
- “属性名称=值”
- “属性名称”

表达式可以是一个等式或者只有属性名称。当它为一个等式时，等式左边不能为空，右边的值可以是值表达式，也可以是被双引号 (") 或单引号 (') 包围的表达式。另外，值可以为空。

下面是一些简单的属性赋值表达式：

```
name="double quoted value"
name='single quoted value'
name=notquotedvaluewithnowhitespace
name
```

属性名称为一个单词，以下正则表达式能够验证属性名称。

```
\w+ (10)
```

以下正则表达式能够验证表达式“属性名称=”（只考虑属性名称和等号，属性的值部分暂时不考虑）。

```
\w+\s*=\s* (11)
```

在正则表达式 `\w+\s*=\s*` 中，`\w+` 匹配属性的名称，`\s*` 匹配任意个空白字符，`=` 匹配赋值语句中的等号 `=`。因此，正则表达式 `\w+\s*=\s*` 可以匹配表达式“属性名称=”（其中，等号两边可以不包含空格或任意个空白字符）。

1. “属性名称=“值””

表达式“属性名称=“值””中的值被双引号包围，但是该值中不能包含双引号。以下正则表达式能够验证表达式““值””。

```
"[^"]*" (12)
```

```
".*?" (13)
```

以下正则表达式能够验证表达式“属性名称=“值””。

```
\w+\s*=\s*"[^"]*" (14)
```

```
\w+\s*=\s*".*?" (15)
```

正则表达式解释如下。

- 正则表达式 `\w+\s*=\s*"[^"]"`：`\w+` 匹配属性名称；`=` 匹配等号；`\s*` 匹配等号两边的空白字符；`"[^"]"` 中的两边的双引号 (") 匹配值两边的双引号，`[^"]` 匹配属性的值。
- 正则表达式 `\w+\s*=\s*".*?"`：`\w+` 匹配属性名称；`=` 匹配等号；`\s*` 匹配等号两边的空白字符；`".*?"` 中的两边的双引号 (") 匹配值两边的双引号，`.*?` 匹配属性的值，它是一个懒惰匹配，将匹配表达式中距第一个双引号最近的双引号。

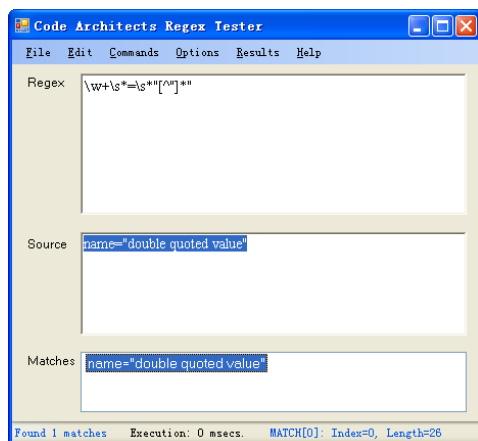
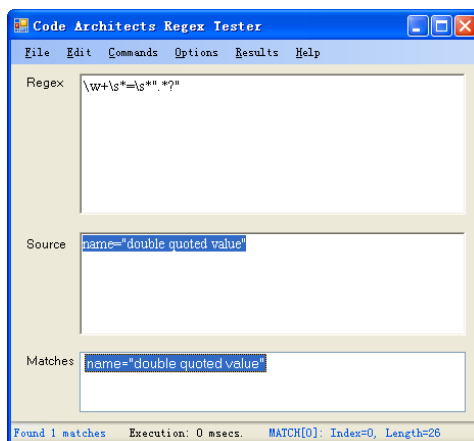
使用工具 **Regex Tester** 分别测试正则表达式 `\w+\s*=\s*"[^"]"` 和 `\w+\s*=\s*".*?"`，结果分别如图 5.7 和图 5.8 所示。

2. “属性名称='值'”

表达式“属性名称='值'”中的值被单引号包围，但是该值中不能包含单引号。以下正则表达式能够验证表达式““值””。

```
'[^']*' (16)
```

```
'.*?' (17)
```

图 5.7 测试正则表达式示例 `\w+\s*=\s*"[^"]*`图 5.8 测试正则表达式 `\w+\s*=\s*".*?"`

以下正则表达式都能够验证表达式“属性名称=值”。

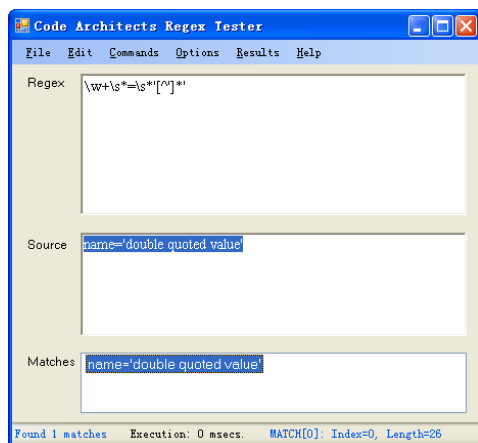
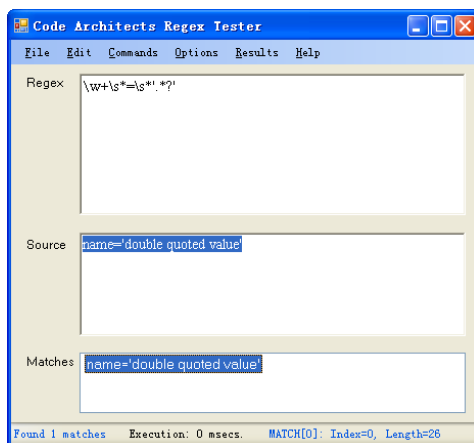
```
\w+\s*=\s*"['"]*" (18)
```

```
\w+\s*=\s*".*?" (19)
```

正则表达式解释如下。

- 正则表达式 `\w+\s*=\s*"['"]*`: `\w+` 匹配属性名称; `=` 匹配等号; `\s*` 匹配等号两边的空白字符; `"['"]*` 中的两边的单引号 (') 匹配值两边的单引号, `[^"]*` 匹配属性的值。
- 正则表达式 `\w+\s*=\s*".*?"`: `\w+` 匹配属性名称; `=` 匹配等号; `\s*` 匹配等号两边的空白字符; `".*?"` 中的两边的单引号 (') 匹配值两边的单引号, `.*` 匹配属性的值, 它是一个懒惰匹配, 将匹配表达式中距第一个单引号最近的单引号。

使用工具 **Regex Tester** 分别测试正则表达式 `\w+\s*=\s*"['"]*` 和 `\w+\s*=\s*".*?"`, 结果分别如图 5.9 和图 5.10 所示。

图 5.9 测试正则表达式示例 `\w+\s*=\s*"['"]*`图 5.10 测试正则表达式 `\w+\s*=\s*".*?"`

3. “属性名称=值”

表达式“属性名称=值”的右边直接是值的表达式。此时, 值表达式中不能包含空白字符。以下正则表达式能够验证表达式“值”。

```
[^ ]+ (20)
```

```
.+?(?= ) (注意, 括号中的表达式的最后是一个空白字符) (21)
```

以下正则表达式能够验证表达式“属性名称=值”。

```
\w+\s*=\s*[^\s]+ (22)
```

```
\w+\s*=\s*.*+?(?=) (23)
```

正则表达式解释如下。

- 正则表达式 `\w+\s*=\s*[^\s]+`: `\w+` 匹配属性名称; `=` 匹配等号; `\s*` 匹配等号两边的空白字符; `[^\s]+` 匹配属性的值, 值表达式中不能包含空白字符。
- 正则表达式 `\w+\s*=\s*.*+?(?=)`: `\w+` 匹配属性名称; `=` 匹配等号; `\s*` 匹配等号两边的空白字符; `.+?(?=)` 中的 `(?=)` 是一个零宽度正预测先行断言, 它将匹配空白字符前面的字符串, `.+?` 是一个懒惰匹配, 将尽可能少地重复所匹配的内容, `.+?(?=)` 匹配表达式中从第一个非空白字符之后的、第一个空白字符之前的字符串。

使用工具 **Regex Tester** 分别测试正则表达式 `\w+\s*=\s*[^\s]+` 和 `\w+\s*=\s*.*+?(?=)`, 结果分别如图 5.11 和图 5.12 所示。

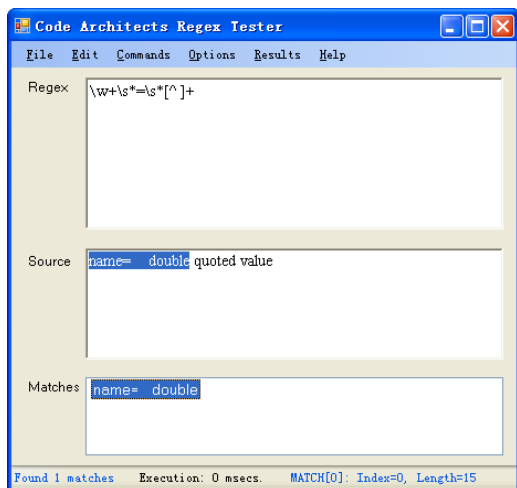


图 5.11 测试正则表达式示例 `\w+\s*=\s*[^\s]+`

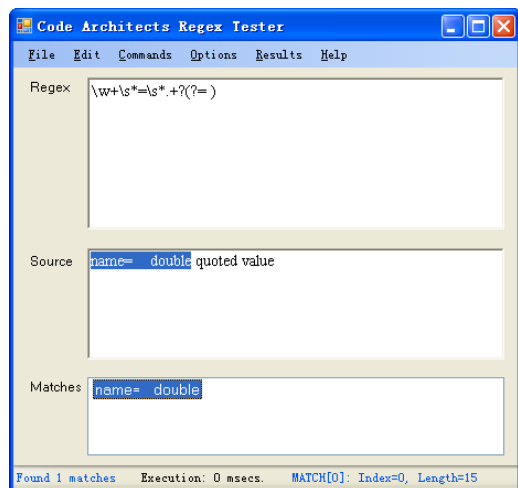


图 5.12 测试正则表达式 `\w+\s*=\s*.*+?(?=)`

4. “属性名称”

匹配表达式“属性名称”, 其实就是匹配属性名称。属性名称为一个单词, 以下正则表达式能够验证属性名称。

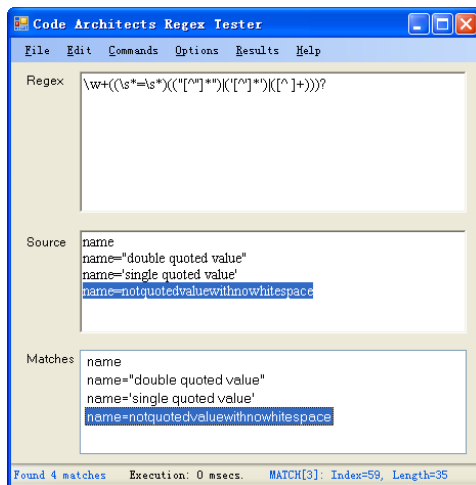
```
\w+ (24)
```

5. HTML 中的属性赋值表达式

综合上述 4 种情况, 以下正则表达式能够验证 HTML 中的属性赋值表达式。

```
\w+((\s*=\s*)(("[^"]*"|'[^']*'|([^\s]+)))?) (25)
```

使用工具 **Regex Tester** 测试正则表达式 `\w+((\s*=\s*)(("[^"]*"|'[^']*'|([^\s]+)))?)`, 结果如图 5.13 所示。

图 5.13 测试正则表达式 `\w+((\s*=s*)(\"[^\"]*\")(('[^']*')|([^\s]+)))?`

5.1.5 HTML 中的注释验证

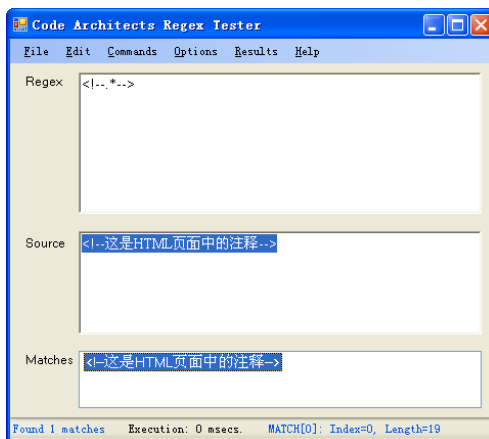
HTML 中的注释一般被符号“<!--”和“-->”包围，如“<!-- 这是该页面中的注释 -->”等。简单验证 HTML 中注释的方法和验证 HTML 标记的方法相似。以下正则表达式能够简单验证 HTML 中的注释。

```
<!--.*--> (26)
```

正则表达式解释如下。

- <!--和-->分别匹配注释的“<!--”和“-->”。
- .*将匹配任意长度的、由非换行字符组成的字符串，即注释的内容。

使用工具 Regex Tester 测试正则表达式 `<!--.*-->`，结果如图 5.14 所示。

图 5.14 测试正则表达式 `<!--.*-->`

正则表达式 `<!--.*-->` 只能简单地验证 HTML 中的注释。如果被匹配的字符串为“<!--这是 HTML 页面-->”中的注释“-->”，则它将匹配整个字符串。然而，如果把字符串“<!--这是 HTML 页面-->”放置在 HTML 页面中，可以发现，实际的注释内容为“<!--这是 HTML 页面-->”，即“中的注释-->”不再是注释。因此，正则表达式 `<!--.*-->` 是一个不精确的验证表达式。

从字符串“<!--这是 HTML 页面-->”中的注释“-->”可以看出，字符串“<!--”把距它最近的字符串“-->”中的内容作为注释。因此，在验证 HTML 中的注释时，正则表达式要找到第一次出现字符串“-->”的位置，这恰恰是一种懒惰匹配。以下正则表达式能够精确验证 HTML 中的注释。

```
<!--.*?-->
```

(27)

正则表达式解释如下。

- <!--和-->分别匹配注释的“<!--”和“-->”。
- .*?将匹配任意长度的、由非换行字符组成的字符串，即注释的内容。但是，该表达式将尽可能少地重复表示的字符，即正则表达式<!--.*?-->中的-->将匹配给定字符串中第一次出现的字符串“-->”（如果存在）。

使用工具 Regex Tester 测试正则表达式<!--.*?-->，结果如图 5.15 所示。

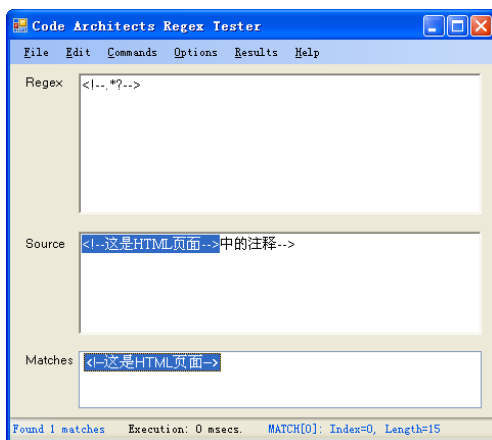


图 5.15 测试正则表达式<!--.*?-->

5.1.6 HTML 中的脚本代码块验证

HTML 中的脚本代码块必须被字符串“<script>”和“</script>”包围，且脚本代码块中可以放置任何内容。以下正则表达式能够简单验证 HTML 中的脚本代码块。

```
<script>.*</script>
```

(28)

正则表达式解释如下。

- <script>、</script>分别匹配注释的“<script>”和“</script>”。
- .*将匹配任意长度的、由非换行字符组成的字符串，即注释的内容。

使用工具 Regex Tester 测试正则表达式<script>.*</script>，结果如图 5.16 所示。

正则表达式<script>.*</script>只能简单地验证 HTML 中的脚本代码块。如果被匹配的字符串为“<script>This is a script block.</script>This is a string.</script>”，则它将匹配整个字符串。然而，如果把字符串“<script>This is a script block.</script>This

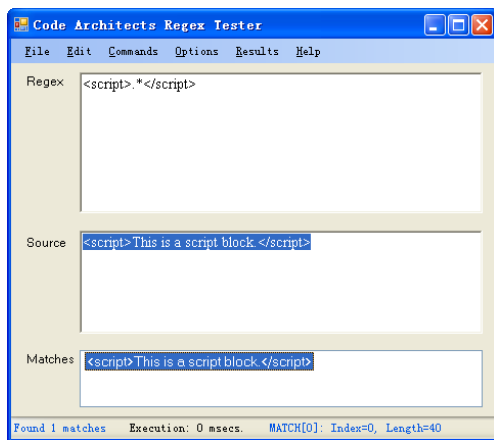


图 5.16 测试正则表达式<script>.*</script>

is a string.</script>”放置在 HTML 页面中，则可以发现，实际的脚本代码块内容为“<script>This is a script block.</script>”，即“<script>This is a string.</script>”不再是脚本代码块。因此，正则表达式<script>.*</script>是一个不精确的验证表达式。

从字符串“<script>This is a script block.</script>This is a string.</script>”可以看出，字符串“<script>”把距它最近的字符串“</script>”中的内容作为脚本代码块。因此，在验证 HTML 中的注释时，正则表达式要找到第一次出现字符串“</script>”的位置。这恰恰是一种懒惰匹配。以下正则表达式能够精确验证 HTML 中的脚本代码块。

```
<script>.*?</script> (29)
```

正则表达式解释如下。

- <script>和</script>分别匹配注释的“<script>”和“</script>”。
- .*?将匹配任意长度的、由非换行字符组成的字符串，即注释的内容。但是，该表达式将尽可能少地重复表示的字符，即正则表达式<script>.*?</script>中的</script>将匹配给定字符串中第一次出现的字符串“</script>”（如果存在）。

使用工具 Regex Tester 测试正则表达式<script>.*?</script>，结果如图 5.17 所示。

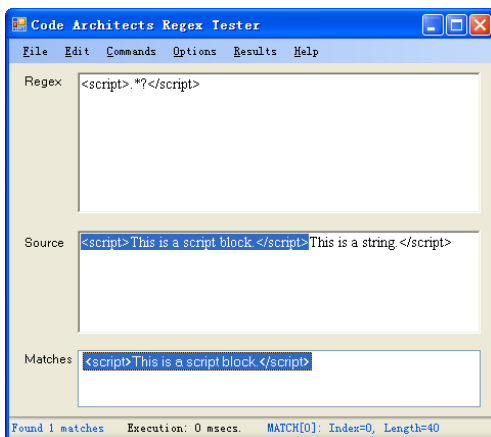


图 5.17 测试正则表达式<script>.*?</script>

5.2 4 种非封闭的 HTML 元素验证

通常，声明 HTML 非封闭元素的语法如下：

```
<tagname 属性赋值表达式[属性赋值表达式...] />
```

其中，声明属性赋值表达式的语法有以下 4 种：

```
attribute="value"
attribute='value'
attribute=value
attribute
```

由 5.1.4 节可知，正则表达式\\w+((\\s*=\\s*)(\"[^\"]*\")|('[^']*')|([^\s]+))?)\"\\s*/>能够验证 HTML 中的属性赋值表达式。以下正则表达式能够简单验证非封闭的 HTML 元素。

```
<\\w+(\\s+\\w+((\\s*=\\s*)(\"[^\"]*\")|('[^']*')|([^\s]+))?)\"\\s*/> (30)
```

正则表达式解释如下。

- \\w+((\\s*=\\s*)(\"[^\"]*\")|('[^']*')|([^\s]+))\"验证一个属性赋值表达式。
- \\s+\\w+((\\s*=\\s*)(\"[^\"]*\")|('[^']*')|([^\s]+))\"验证以空白字符开头的、后接一个属性赋值表

达式。

- ❑ `(\s+\w+(\s*=\s*("[^"]*"|'['']*'|([^\s]+)))?)*` 可用来验证 0 个、1 个或多个由空白字符和属性赋值表达式组成的字符串。
- ❑ `<\w+` 中的 `<` 验证元素的开头字符 `<`, `\w` 验证元素的名称。
- ❑ `\s*/>` 验证元素的结束标记及其前面的空白字符（如果存在）。

使用工具 **Regex Tester** 测试正则表达式 `<\w+(\s+\w+(\s*=\s*("[^"]*"|'['']*'|([^\s]+)))?)*\s*/>`, 结果如图 5.18 所示。

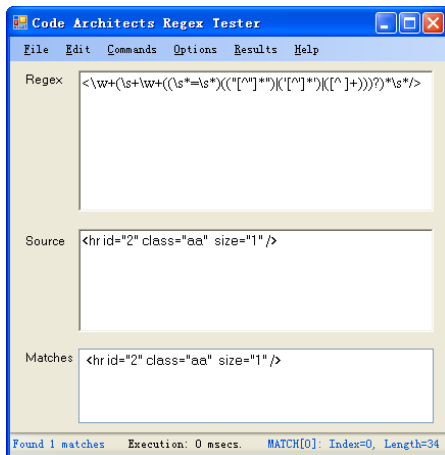


图 5.18 测试正则表达式 `<\w+(\s+\w+(\s*=\s*("[^"]*"|'['']*'|([^\s]+)))?)*\s*/>`

5.2.1
元素验证

`
` 元素是一个非封闭的 HTML 元素，声明该控件的语法如下：

```
<br 属性赋值表达式 [属性赋值表达式...] />
```

其中，声明属性赋值表达式的语法有以下几种：

```
attribute="value"
attribute='value'
attribute=value
attribute
```

由 5.1.4 节可知，正则表达式 `\w+(\s*=\s*("[^"]*"|'['']*'|([^\s]+)))?` 能够验证 HTML 中的属性赋值表达式。以下正则表达式能够简单验证 `
` 元素。

```
<br (\s+\w+ ( (\s*=\s*) ( ("[""]*" | ('['']*'| ([^\s]+) ) ) ) ) ? ) * \s*/> (31)
```

正则表达式解释如下。

- ❑ `\w+(\s*=\s*("[^"]*"|'['']*'|([^\s]+)))?` 验证一个属性赋值表达式。
- ❑ `\s+\w+(\s*=\s*("[^"]*"|'['']*'|([^\s]+)))?` 验证以空白字符开头的、后接一个属性赋值表达式。
- ❑ `(\s+\w+(\s*=\s*("[^"]*"|'['']*'|([^\s]+)))?)*` 可用来验证 0 个、1 个或多个由空白字符和属性赋值表达式组成的字符串。
- ❑ `<br` 验证 `
` 元素的开头 3 个字符。
- ❑ `\s*/>` 验证 `
` 元素的结束标记及其前面的空白字符（如果存在）。

使用工具 **Regex Tester** 测试正则表达式 `<br(\s+\w+(\s*=\s*("[^"]*"|'['']*'|([^\s]+)))?)*\s*/>`, 结果如图 5.19 所示。

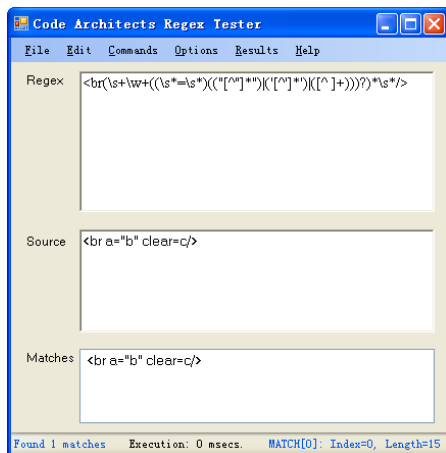


图 5.19 测试正则表达式<br(\s+\w+(\s*=\s*("[^"]*"|'['']*'|([^\s]+)))?\s*/>

元素只包含以下 5 个属性。

- ❑ id: 元素的 ID 值，它的值为一个字符串。
- ❑ class: 样式表中的类名称，它的值为一个字符串。
- ❑ title: 元素的标题，它的值为一个字符串。
- ❑ style: 元素的呈现风格。
- ❑ clear: 元素的显示位置，它的值为 left 或者 right。

由 5.1.4 节可知，正则表达式 `\w+(\s*=\s*("[^"]*"|'['']*'|([^\s]+)))?` 能够验证 HTML 中的属性赋值表达式。以下正则表达式能够验证属性 id 的赋值表达式。

```
id\s*=\s*(("[^"]*"|'['']*'|([^\s]+)))
```

 (32)

以下正则表达式能够验证属性 class 的赋值表达式。

```
class\s*=\s*(("[^"]*"|'['']*'|([^\s]+)))
```

 (33)

以下正则表达式能够验证属性 title 的赋值表达式。

```
title\s*=\s*(("[^"]*"|'['']*'|([^\s]+)))
```

 (34)

以下正则表达式能够验证属性 style 的赋值表达式。

```
style\s*=\s*(("[^"]*"|'['']*'|([^\s]+)))
```

 (35)

以下正则表达式能够验证属性 clear 的赋值表达式。

```
clear\s*=\s*(?<char>["']?)(left|right)\k<char>
```

 (36)

正则表达式解释如下。

- ❑ 分组 `(?<char>["']?)` 可用来验证空字符，或者验证字符“或”，并将该分组验证的内容命名为“char”。
- ❑ `(left)|(right)` 匹配字符串“left”或者“right”。
- ❑ `\k<char>` 匹配名称为“char”的分组内容。
- ❑ `(?<char>["']?)(left)|(right)\k<char>` 可用来匹配“"left"”、“'left'”、“left”、“"right"”、“'right'”或“right”。

综上所述，以下正则表达式能够精确验证
元素。

```
<br
\s+id\s*=\s*(("[^"]*"|'['']*'|([^\s]+)))?
\s+class\s*=\s*(("[^"]*"|'['']*'|([^\s]+)))?
\s+title\s*=\s*(("[^"]*"|'['']*'|([^\s]+)))?
\s+style\s*=\s*(("[^"]*"|'['']*'|([^\s]+)))?
```



```
(\s+clear\s*=\s*(?<char>["' ]?)((left)|(right))\k<char>)?
\s*/>
```

(37)

正则表达式解释如下。

- `(\s+id\s*=\s*((("[^"]*"|('''))|([^\s]+)))`?验证以空白字符开头的、后接属性 `id` 的赋值表达式（如果存在）。
- `(\s+class\s*=\s*((("[^"]*"|('''))|([^\s]+)))`?验证以空白字符开头的、后接属性 `class` 的赋值表达式（如果存在）。
- `(\s+title\s*=\s*((("[^"]*"|('''))|([^\s]+)))`?验证以空白字符开头的、后接属性 `title` 的赋值表达式（如果存在）。
- `(\s+style\s*=\s*((("[^"]*"|('''))|([^\s]+)))`?验证以空白字符开头的、后接属性 `style` 的赋值表达式（如果存在）。
- `(\s+clear\s*=\s*(?<char>["']?)((left)|(right))\k<char>)?`验证以空白字符开头的、后接属性 `clear` 的赋值表达式（如果存在）。
- `<br` 验证 `
` 元素的开头 3 个字符。
- `\s*/>` 验证 `
` 元素的结束标记及其前面的空白字符（如果存在）。

使用工具 **Regex Tester** 测试正则表达式示例 (37)，结果如图 5.20 所示。

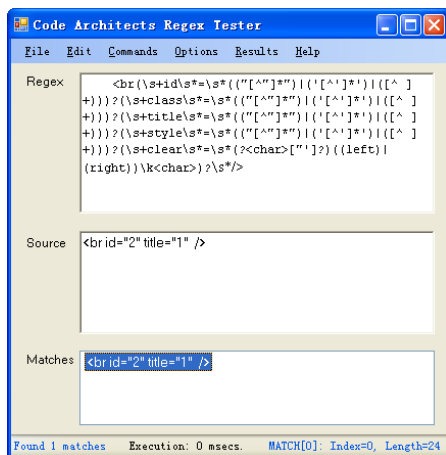


图 5.20 测试正则表达式示例 (37)

注意：能够被示例 (37) 验证的 `
` 元素的属性 `id`、`class`、`title`、`style` 和 `clear` 的赋值表达式的出现顺序是固定的，即按照从左到右依次为 `id`、`class`、`title`、`style`、`clear` 的这样一个顺序。读者可以根据自己的需要进行调整。

5.2.2 <hr>元素验证

`<hr>` 元素是一个非封闭的 HTML 元素，声明该控件的语法如下。

```
<hr 属性赋值表达式[属性赋值表达式...] />
```

其中，声明属性赋值表达式的语法有以下 4 种。

```
attribute="value"
attribute='value'
attribute=value
attribute
```

由 5.1.4 节可知，正则表达式 `w+((\s*=\s*((("[^"]*"|('''))|([^\s]+)))`?能够验证 HTML 中的属性

赋值表达式。以下正则表达式能够验证<hr>元素。

```
<hr(\s+\w+((\s*=\s*)(("[^"]*"|'['']*'|([^\s]+)))?)*\s*/> (38)
```

正则表达式解释如下。

- ❑ \w+((\s*=\s*)(("[^"]*"|'['']*'|([^\s]+)))?)验证一个属性赋值表达式。
- ❑ \s+\w+((\s*=\s*)(("[^"]*"|'['']*'|([^\s]+)))?)验证以空白字符开头的、后接一个属性赋值表达式。
- ❑ (\s+\w+((\s*=\s*)(("[^"]*"|'['']*'|([^\s]+)))?))*用来验证 0 个、1 个或多个由空白字符和属性赋值表达式组成的字符串。
- ❑ <hr 验证<hr>元素的开头 3 个字符。
- ❑ \s*/>验证<hr>元素的结束标记及其前面的空白字符（如果存在）。

使用工具 **Regex Tester** 测试正则表达式<hr(\s+\w+((\s*=\s*)(("[^"]*"|'['']*'|([^\s]+)))?)*\s*/>，结果如图 5.21 所示。

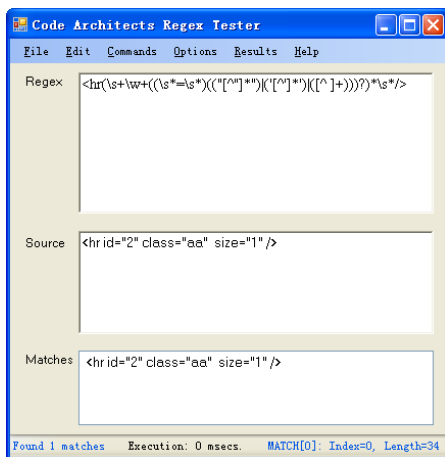


图 5.21 测试正则表达式<hr(\s+\w+((\s*=\s*)(("[^"]*"|'['']*'|([^\s]+)))?)*\s*/>

<hr>元素包含以下多个属性。

- ❑ id: 元素的 ID 值，它的值为一个字符串。
- ❑ class: 样式表中的类名称，它的值为一个字符串。
- ❑ dir: 元素的描述，它的值为一个字符串。
- ❑ lang: 元素与语言相关的信息。
- ❑ style: 元素的呈现风格。
- ❑ align: 元素的显示位置，它的值可为 left、center 和 right。
- ❑ noshade: 表示元素是否有阴影。
- ❑ size: 元素的大小。
- ❑ width: 元素的宽度，它的值可以是整数或者百分比。
- ❑ onxxxxxx: 元素的事件，它的值为一个字符串。

由 5.1.4 节可知，正则表达式 \w+((\s*=\s*)(("[^"]*"|'['']*'|([^\s]+)))?) 能够验证 HTML 中的属性赋值表达式。以下正则表达式能够验证属性 id 的赋值表达式。

```
id\s*=\s*((("[^"]*"|'['']*'|([^\s]+))) (39)
```

以下正则表达式能够验证属性 class 的赋值表达式。

```
class\s*=\s*((("[^"]*"|'['']*'|([^\s]+))) (40)
```

以下正则表达式能够验证属性 **dir** 的赋值表达式。

```
dir\s*=\s*(("[^"]*"|'['']*'|([^\s]+))
```

 (41)

以下正则表达式能够验证属性 **lang** 的赋值表达式。

```
lang\s*=\s*(("[^"]*"|'['']*'|([^\s]+))
```

 (42)

以下正则表达式能够验证属性 **style** 的赋值表达式。

```
style\s*=\s*("[^"]*"|'['']*'|([^\s]+))
```

 (43)

以下正则表达式能够验证属性 **align** 的赋值表达式。

```
align\s*=\s*(?<char>["']?)(left)|(center)|(right))\k<char>
```

 (44)

以下正则表达式能够验证属性 **noshade** 的赋值表达式。

```
noshade
```

 (45)

以下正则表达式能够验证属性 **size** 的赋值表达式。

```
size\s*=\s*"(\d|([1-9]\d*))"
```

 (46)

以下正则表达式能够验证属性 **width** 的赋值表达式。

```
width\s*=\s*"(\d|([1-9]\d*))%?"
```

 (47)

以下正则表达式能够验证属性 **dir** 的赋值表达式。

```
on\w+\s*=\s*("[^"]*"|'['']*'|([^\s]+))
```

 (48)

正则表达式解释如下。

- ❑ 分组(?<char>["']?)可用来验证空字符，或者验证字符"或'，并将该分组验证的内容命名为“char”。
- ❑ (left)|(center)|(right)匹配字符串“left”、“right”或者“center”。
- ❑ \k<char>匹配名称为“char”的分组内容。
- ❑ (?<char>["']?)(left)|(center)|(right))\k<char>可用来匹配“"left”、“'left”、“left”、“"right”、“'right”、“right”、“"center”、“'center”或“center”。
- ❑ \d|([1-9]\d*匹配以非 0 开头的整数。
- ❑ (\d|([1-9]\d*))%?匹配以非 0 开头的整数，或者百分比。

综上所述，以下正则表达式能够精确验证<hr>元素。

```
<hr
(\s+id\s*=\s*("[^"]*"|'['']*'|([^\s]+)))?
(\s+class\s*=\s*("[^"]*"|'['']*'|([^\s]+)))?
(\s+dir\s*=\s*("[^"]*"|'['']*'|([^\s]+)))?
(\s+lang\s*=\s*("[^"]*"|'['']*'|([^\s]+)))?
(\s+style\s*=\s*("[^"]*"|'['']*'|([^\s]+)))?
(\s+align\s*=\s*(?<char>["']?)(left)|(center)|(right))\k<char>)?
(\s+noshade)?
(\s+size\s*=\s*"(\d|([1-9]\d*))")?
(\s+width\s*=\s*"(\d|([1-9]\d*))%?" )?
(\s+on\w+\s*=\s*("[^"]*"|'['']*'|([^\s]+)))?
\s*/>
```

 (49)

正则表达式解释如下。

- ❑ (\s+id\s*=\s*("[^"]*"|'['']*'|([^\s]+)))?验证以空白字符开头的、后接属性 **id** 的赋值表达式（如果存在）。
- ❑ (\s+class\s*=\s*("[^"]*"|'['']*'|([^\s]+)))?验证以空白字符开头的、后接属性 **class** 的赋值表达式（如果存在）。
- ❑ (\s+dir\s*=\s*("[^"]*"|'['']*'|([^\s]+)))?验证以空白字符开头的、后接属性 **title** 的赋值表达式（如果存在）。

- ❑ `(\s+style\s*=\s*(("[^"]*"|('[^']*'|([^\s]+)))?)`验证以空白字符开头的、后接属性 `style` 的赋值表达式（如果存在）。
- ❑ `(\s+align\s*=\s*(?<char>["']?)(left)|(center)|(right))\k<char>?)`验证以空白字符开头的、后接属性 `clear` 的赋值表达式（如果存在）。
- ❑ `(\s+noshade)?`验证以空白字符开头的、后接属性 `noshade` 的表达式。
- ❑ `(\s+size\s*=\s*"(\d{1-9}\d*)")`验证以空白字符开头的、后接属性 `size` 的赋值表达式（如果存在）。
- ❑ `(\s+width\s*=\s*"(\d{1-9}\d*)%")`验证以空白字符开头的、后接属性 `width` 的赋值表达式（如果存在）。
- ❑ `<hr` 验证 `<hr>` 元素的开头三个字符。
- ❑ `\s*/>` 验证 `<hr>` 元素的结束标记及其前面的空白字符（如果存在）。

使用工具 `Regex Tester` 测试正则表达式示例（49），结果如图 5.22 所示。

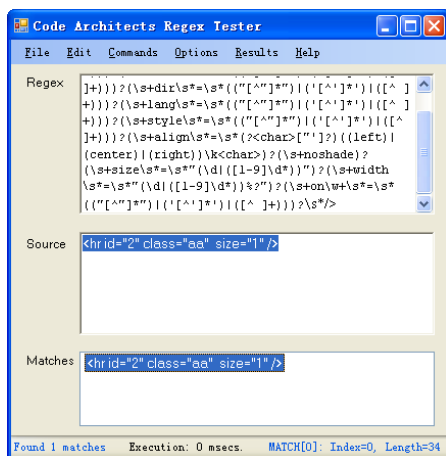


图 5.22 测试正则表达式示例（49）

注意：和示例（37）一样，能够被示例（49）验证的 `<hr>` 元素的属性也存在着出现顺序的问题。读者可以根据自己的需要进行调整。

5.2.3 <a>元素验证

`<a>` 元素是一个封闭的 HTML 元素，声明该控件的语法如下：

`<a 属性赋值表达式[属性赋值表达式...] >`嵌入的内容`<a>`

其中，声明属性赋值表达式的语法有以下 4 种。

```
attribute="value"
attribute='value'
attribute=value
attribute
```

根据 5.1.4 节可知，正则表达式 `\w+((\s*=\s*("[^"]*"|('[^']*'|([^\s]+)))?)` 能够验证 HTML 中的属性赋值表达式。以下正则表达式能够简单验证 `<a>` 元素。

```
<a(\s+(\s*=\s*("[^"]*"|('[^']*'|([^\s]+)))?)\s*.*?</a> (50)
```

正则表达式解释如下。

- ❑ `\w+((\s*=\s*("[^"]*"|('[^']*'|([^\s]+)))?)`验证一个属性赋值表达式。
- ❑ `\s+(\s*=\s*("[^"]*"|('[^']*'|([^\s]+)))?)`验证以空白字符开头的、后接一个属性赋值表

达式。

- ❑ `(\s+\w+((\s*=\s*)(("[^"]*"|(''')|([^\]+)))?)*\s*>` 可用验证 0 个、1 个或多个由空白字符和属性赋值表达式组成的字符串。
- ❑ `<a` 验证 `<a>` 元素开头的两个字符。
- ❑ `\s*>` 验证元素开始标记中的结束字符 `>`。
- ❑ `.?*` 验证元素的开始标记和结束标记之间的嵌入内容。
- ❑ `` 匹配 `<a>` 元素的结束标记。

使用工具 **Regex Tester** 测试正则表达式 `<a(\s+\w+((\s*=\s*)(("[^"]*"|(''')|([^\]+)))?)*\s*>.*?\s*`, 结果如图 5.23 所示。

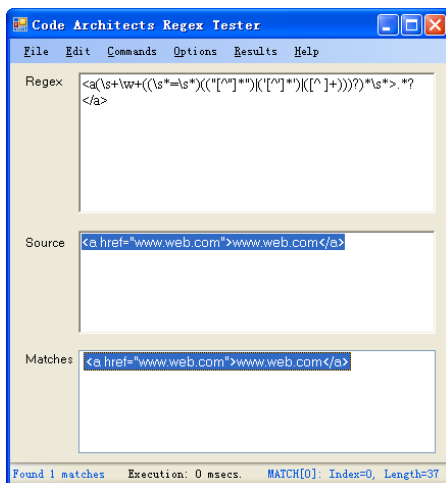


图 5.23 测试正则表达式 `<a(\s+\w+((\s*=\s*)(("[^"]*"|(''')|([^\]+)))?)*\s*>.*?\s*`

`<a>` 元素包含以下多个属性。

- ❑ **accesskey**: 元素的访问键, 它的值为一个字母字符。
- ❑ **charset**: 编码, 它的值为一个字符串。
- ❑ **class**: 样式表中的类名称, 它的值为一个字符串。
- ❑ **coords**: 与元素相关的 **map** 的形状描述, 它的值可以是 **rect**、**circle** 和 **poly**。
- ❑ **dir**: 元素的描述, 它的值为一个字符串。
- ❑ **href**: 元素的链接地址, 它的值为一个字符串。
- ❑ **hreflang**: 与元素的链接地址相关的语言, 它的值为一个字符串。
- ❑ **id**: 元素的 ID 值, 它的值为一个字符串。
- ❑ **lang**: 元素与语言相关的信息, 它的值为一个字符。
- ❑ **name**: 元素的名称, 一般以字符 “#” 开头, 它的值为一个字符串。
- ❑ **onxxxxxx**: 元素的事件, 它的值为一个字符串。
- ❑ **rel**: 链接的类型, 它的值为一个字符串。
- ❑ **rev**: 链接的类型, 它的值为一个字符串。
- ❑ **shape**: 与元素相关的 **map** 的形状描述, 它的值可以是 **default**、**rect**、**circle** 和 **poly**。
- ❑ **style**: 元素的呈现风格, 它的值为一个字符。
- ❑ **tabindex**: 元素与 **Tab** 键相关的索引, 它的值为一个整数。

- **target**: 元素的目标窗口或框架，它的值是一个字符串。
- **title**: 元素的标题，它的值为一个字符串。
- **type**: 目标地址的 Content-Type 属性值，它的值为一个字符串。

由 5.1.4 节可知，正则表达式 `w+(\s*=\s*("[^"]*"|'[^']*'|([^\s]*)+))?` 能够验证 HTML 中的属性赋值表达式。以下正则表达式能够验证属性 **accesskey** 的赋值表达式。

```
accesskey\s*=\s*[a-zA-Z] (51)
```

以下正则表达式能够验证属性 **charset** 的赋值表达式。

```
charset\s*=\s*("[^"]*"|'[^']*'|([^\s]*)+) (52)
```

以下正则表达式能够验证属性 **class** 的赋值表达式。

```
class\s*=\s*("[^"]*"|'[^']*'|([^\s]*)+) (53)
```

以下正则表达式能够验证属性 **coords** 的赋值表达式。

```
coords\s*=\s*(?<char>["']?)((rect)|(circle)|(poly))\k<char> (54)
```

以下正则表达式能够验证属性 **dir** 的赋值表达式。

```
dir\s*=\s*("[^"]*"|'[^']*'|([^\s]*)+) (55)
```

以下正则表达式能够验证属性 **href** 的赋值表达式。

```
href\s*=\s*("[^"]*"|'[^']*'|([^\s]*)+) (56)
```

以下正则表达式能够验证属性 **hreflang** 的赋值表达式。

```
hreflang\s*=\s*("[^"]*"|'[^']*'|([^\s]*)+) (57)
```

以下正则表达式能够验证属性 **id** 的赋值表达式。

```
id\s*=\s*("[^"]*"|'[^']*'|([^\s]*)+) (58)
```

以下正则表达式能够验证属性 **lang** 的赋值表达式。

```
lang\s*=\s*("[^"]*"|'[^']*'|([^\s]*)+) (59)
```

以下正则表达式能够验证属性 **name** 的赋值表达式。

```
name\s*=\s*("[^"]*"|'[^']*'|(#[^\s]*)+) (60)
```

以下正则表达式能够验证属性 **onxxxxxx**（元素的事件）的赋值表达式。

```
onw+\s*=\s*("[^"]*"|'[^']*'|([^\s]*)+) (61)
```

以下正则表达式能够验证属性 **rel** 的赋值表达式。

```
rel\s*=\s*("[^"]*"|'[^']*'|([^\s]*)+) (62)
```

以下正则表达式能够验证属性 **rev** 的赋值表达式。

```
rev\s*=\s*("[^"]*"|'[^']*'|([^\s]*)+) (63)
```

以下正则表达式能够验证属性 **shape** 的赋值表达式。

```
shape\s*=\s*(?<char>["']?)((default)|(rect)|(circle)|(poly))\k<char> (64)
```

以下正则表达式能够验证属性 **style** 的赋值表达式。

```
style\s*=\s*("[^"]*"|'[^']*'|([^\s]*)+) (65)
```

以下正则表达式能够验证属性 **tabindex** 的赋值表达式。

```
tabindex\s*=\s*"((-1)|\d|([1-9]\d*))" (66)
```

以下正则表达式能够验证属性 **target** 的赋值表达式。

```
target\s*=\s*("[^"]*"|'[^']*'|([^\s]*)+) (67)
```

以下正则表达式能够验证属性 **title** 的赋值表达式。

```
title\s*=\s*("[^"]*"|'[^']*'|([^\s]*)+) (68)
```

以下正则表达式能够验证属性 **type** 的赋值表达式。

```
type\s*=\s*("[^"]*"|'[^']*'|([^\s]*)+) (69)
```

正则表达式解释如下。

- `[a-zA-Z]` 可以匹配一个字母，`accesskey\s*=\s*[a-zA-Z]` 匹配属性 **accesskey** 的赋值表达式。

- ❑ 分组(?<char>["']?)可用来验证空字符，或者验证字符"或'，并将该分组验证的内容命名为“char”。
- ❑ (rect)|(circle)|(poly)匹配字符串“rect”、“circle”或者“poly”。
- ❑ \k<char>匹配名称为“char”的分组内容。
- ❑ coords\s*=\s*(?<char>["']?)(rect)|(circle)|(poly))\k<char>匹配属性 coords 的赋值表达式。
- ❑ name\s*=\s*((#[^"]*)|('[^']*')|([^\s]+))匹配属性 name 的赋值表达式。其中，值必须以字符“#”开头。
- ❑ onw\s*=\s*("[^"]*"|'['']*'|([^\s]+))匹配属性 onxxxxxx (元素的事件) 的赋值表达式。其中，属性名称必须以字符串“on”开头。
- ❑ tabindex\s*=\s*((-1)|d|([1-9]\d*))匹配属性 tabindex 的赋值表达式。其中，表达式 (-1)|d|([1-9]\d*)可以匹配-1、一位整数或者非0开头的任意整数。

综上所述，以下正则表达式能够精确验证<a>元素。

```
<a
(\s+accesskey\s*=\s*[a-zA-Z])?
(\s+charset\s*=\s*("[^"]*"|'['']*'|([^\s]+)))?
(\s+class\s*=\s*("[^"]*"|'['']*'|([^\s]+)))?
(\s+coords\s*=\s*(?<char>["']?)(rect)|(circle)|(poly))\k<char>)?
(\s+dir\s*=\s*("[^"]*"|'['']*'|([^\s]+)))?
(\s+href\s*=\s*("[^"]*"|'['']*'|([^\s]+)))?
(\s+hreflang\s*=\s*("[^"]*"|'['']*'|([^\s]+)))?
(\s+id\s*=\s*("[^"]*"|'['']*'|([^\s]+)))?
(\s+lang\s*=\s*("[^"]*"|'['']*'|([^\s]+)))?
(\s+name\s*=\s*((#[^"]*)|'['']*'|([^\s]+)))?
(\s+onw\s*=\s*("[^"]*"|'['']*'|([^\s]+)))?
(\s+ rel\s*=\s*("[^"]*"|'['']*'|([^\s]+)))?
(\s+ rev\s*=\s*("[^"]*"|'['']*'|([^\s]+)))?
(\s+shape\s*=\s*(?<schar>["']?)(default)|(rect)|(circle)|(poly))\k<schar>)?
(\s+style\s*=\s*("[^"]*"|'['']*'|([^\s]+)))?
(\s+tabindex\s*=\s*((-1)|d|([1-9]\d*))"?
(\s+target\s*=\s*("[^"]*"|'['']*'|([^\s]+)))?
(\s+title\s*=\s*("[^"]*"|'['']*'|([^\s]+)))?
\s*>.*?</a>
```

(70)

正则表达式解释如下。

- ❑ (\s+id\s*=\s*("[^"]*"|'['']*'|([^\s]+)))?验证以空白字符开头的、后接属性 id 的赋值表达式（如果存在）。
- ❑ (\s+class\s*=\s*("[^"]*"|'['']*'|([^\s]+)))?验证以空白字符开头的、后接属性 class 的赋值表达式（如果存在）。
- ❑ (\s+style\s*=\s*("[^"]*"|'['']*'|([^\s]+)))?验证以空白字符开头的、后接属性 style 的赋值表达式（如果存在）。
- ❑ <a 验证<a>元素开头的两个字符。
- ❑ \s*>验证元素开始标记中的结束字符>。
- ❑ .*?验证元素的开始标记和结束标记之间的嵌入内容。
- ❑ 匹配<a>元素的结束标记。

使用工具 Regex Tester 测试正则表达式示例（70），结果如图 5.24 所示。

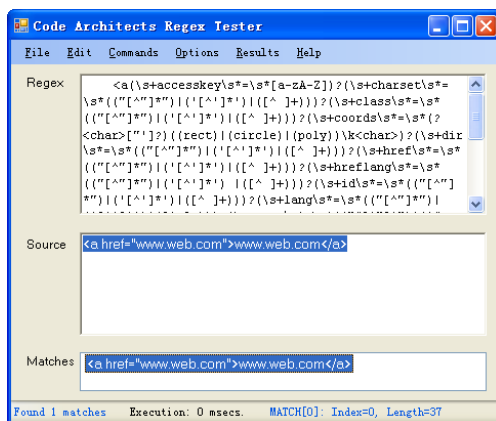


图 5.24 测试正则表达式示例（70）

注意：和示例（37）一样，能够被示例（70）验证的<a>元素的属性也存在着出现顺序的问题。读者可以根据自己的需要进行调整。

5.2.4 <input>元素验证

<input>元素是一个非封闭的 HTML 元素，声明该控件的语法如下。

```
<input 属性赋值表达式[属性赋值表达式...] />
```

其中，声明属性赋值表达式的语法有以下几种。

```
attribute="value"
attribute='value'
attribute=value
attribute
```

由 5.1.4 节可知，正则表达式`\w+(\s*=\s*)(("[^"]*"|'['']*'|([^\s]+)))?`能够验证 HTML 中的属性赋值表达式。以下正则表达式能够简单验证<input>元素。

```
<input (\s+\w+(\s*=\s*)(("[^"]*"|'['']*'|([^\s]+)))?) * \s*/> (71)
```

正则表达式解释如下。

- ❑ `\w+(\s*=\s*)(("[^"]*"|'['']*'|([^\s]+)))?`验证一个属性的赋值表达式。
- ❑ `\s+\w+(\s*=\s*)(("[^"]*"|'['']*'|([^\s]+)))?`验证以空白字符开头的、后接一个属性的赋值表达式。
- ❑ `(\s+\w+(\s*=\s*)(("[^"]*"|'['']*'|([^\s]+)))?) *`可用来验证 0 个、1 个或多个由空白字符和属性的赋值表达式组成的字符串。
- ❑ `<input` 验证<input>元素开头的两个字符。
- ❑ `\s*/>`验证元素的结束标记及其前面的空白字符（如果存在）。

使用工具 Regex Tester 测试正则表达式`<input(\s+\w+(\s*=\s*)(("[^"]*"|'['']*'|([^\s]+)))?) * \s*/>`，结果如图 5.25 所示。

<input>元素存在一个必需的属性 type，它标识元素的类型。<input>元素根据属性 type 的值呈现相应的控件。属性 type 的值可以是以下 10 种。

- ❑ text，显示一个文本输入框。
- ❑ password，显示一个密码输入框。
- ❑ checkbox，显示一个复选框。
- ❑ radio，显示一个单选框。

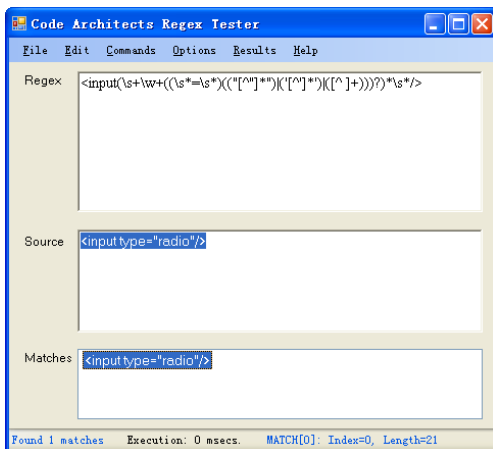


图 5.25 测试正则表达式 `<input(\s+\w+(\s*=\s*)(("[^"]*"|'['']*|([^\s]+)))?\s*/>`

- ❑ submit, 显示一个提交按钮。
- ❑ reset, 显示一个重置按钮。
- ❑ file, 显示一个浏览文件按钮。
- ❑ hidden, 隐藏控件。
- ❑ image, 图像控件。
- ❑ button, 普通按钮控件。

以下正则表达式能够验证 `<input>` 元素属性 `type` 的赋值表达式。

```
type\s*=\s*((text)|(password)|(checkbox)|(radio)|(submit)|(reset)|(file)|(hidden)|(image)|(button))
```

(72)

5.3 封闭的 HTML 元素验证

通常, 声明 HTML 封闭元素的语法如下:

```
<tagname 属性赋值表达式[属性赋值表达式...] >HTML 文本</tagname>
```

其中, 声明属性赋值表达式的语法有以下 4 种。

```
attribute="value"
attribute='value'
attribute=value
attribute
```

由 5.1.4 节可知, 正则表达式 `\w+(\s*=\s*)(("[^"]*"|'['']*|([^\s]+)))?` 能够验证 HTML 中的属性赋值表达式。以下正则表达式能够简单验证非封闭的 HTML 元素。

```
<(?(<name>\w+) (\s+\w+(\s*=\s*) ("["[^"]*"|'["']*'|(["^\s]+)))?) * \s*>.*?</k<name>>
```

(73)

正则表达式解释如下。

- ❑ `\w+(\s*=\s*)(("[^"]*"|'['']*|([^\s]+)))?` 验证一个属性赋值表达式。
- ❑ `\s+\w+(\s*=\s*)(("[^"]*"|'['']*|([^\s]+)))?` 验证以空白字符开头的、后接一个属性的赋值表达式。
- ❑ `(\s+\w+(\s*=\s*)(("[^"]*"|'['']*|([^\s]+)))?)*` 可用来验证 0 个、1 个或多个由空白字符和属性赋值表达式组成的字符串。
- ❑ `<(?(<name>\w+) >` 验证元素的开头字符 `<`; `(?(<name>\w+) >` 验证元素的名称, 并将验证

的内容保存在名称为“name”的分组中。

- ❑ \s*>验证元素的开始标记中的结束字符>。
- ❑ .*?验证元素的开始标记和结束标记之间的内容。
- ❑ </k<name>>匹配元素的结束标记，并且使用名称为“name”的分组所匹配的内容。

使用工具 Regex Tester 测试正则表达式示例（73），结果如图 5.26 所示。

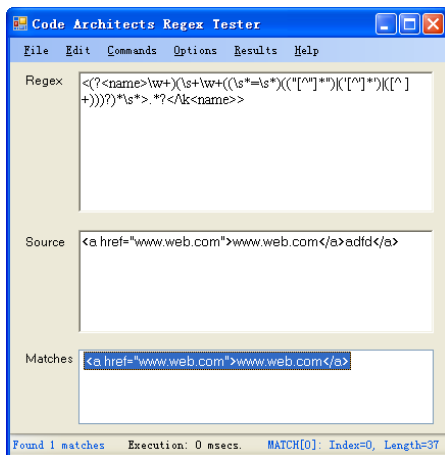


图 5.26 测试正则表达式示例（73）

5.4 处理 HTML 元素

本节将介绍使用正则表达式处理 HTML 元素的方法。如提取 HTML 标记、提取 HTML 标记之间的内容、提取 URL、提取图像的 URL 和提取 HTML 页面的标题等。

5.4.1 提取 HTML 标记

HTML 标记一般被尖括号包围，如<a>、<table>、
、<input>等。以下的正则表达式能够从 HTML 文本中提取所有 HTML 标记。

```
</?[a-zA-Z][^>]*> (74)
```

正则表达式解释如下。

- ❑ <匹配 HTML 标记的左尖括号。
- ❑ /?匹配空字符或者字符/。如果匹配空字符，则匹配 HTML 标记的开头部分，否则匹配 HTML 标记的结尾部分。
- ❑ 字符类[a-zA-Z]可以匹配一个英文字母，它匹配 HTML 标记中第一个字符（除去左尖括号）。
- ❑ 字符类[^>]可以匹配除右尖括号的之外的任意字符。
- ❑ [^>]*可以匹配空字符串，或者除右尖括号的之外的任意字符组成的字符串。
- ❑ >匹配 HTML 标记的右尖括号。
- ❑ [a-zA-Z][^>]*匹配 HTML 标记的名称。

使用工具 Regex Tester 测试正则表达式</?[a-zA-Z][^>]*>，结果如图 5.27 所示。

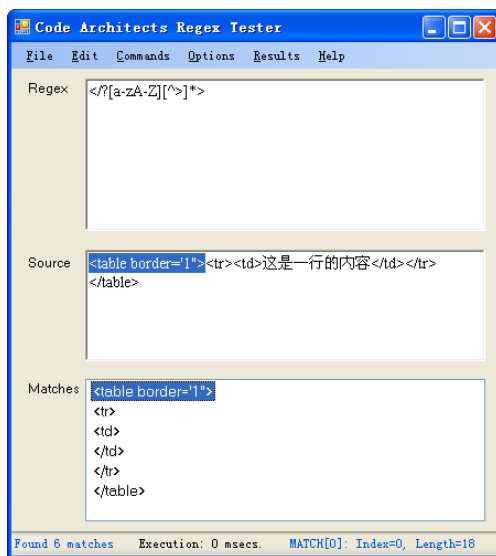


图 5.27 测试正则表达式<?/[a-zA-Z][^>]*>

5.4.2 提取 HTML 标记之间的内容

HTML 标记之间的内容一般被 HTML 标记的开头部分和结尾部分包围。不妨设给定的 HTML 文本如下：

```
<td>这是一行的内容</td>
```

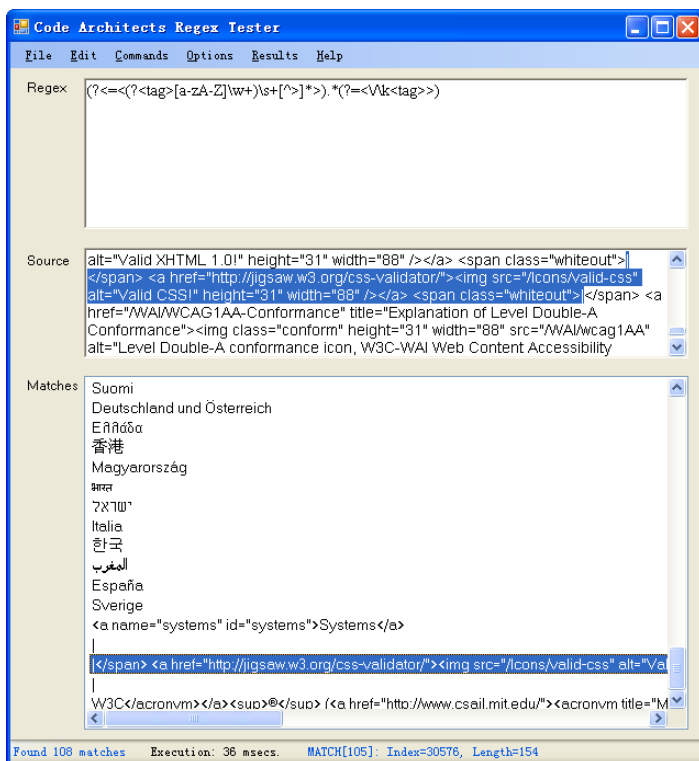
那么，HTML 标记之间的内容为“这是一行的内容”。以下正则表达式能够匹配 HTML 标记之间的内容。

```
(?<=<(?(tag>[a-zA-Z]\w+)\s+[>]*)).*?(?=<\k<tag>>) (75)
```

正则表达式解释如下。

- `(?(tag>[a-zA-Z]\w+)\s+[>]*)` 是一个分组，它的名称为“tag”。在该正则表达式匹配过程中，它将保存被匹配的内容。
- `<(?(tag>[a-zA-Z]\w+)\s+[>]*)` 匹配 HTML 标记的开头标记，如 `<a>`、`
`、`<hr>`、`<table border="1">` 等。
- `(?<=<(?(tag>[a-zA-Z]\w+)\s+[>]*)` 是一个零宽度正回顾后发断言，它断言自身位置的前面能够匹配 `<(?(tag>[a-zA-Z]\w+)\s+[>]*)` 所匹配的内容。在此，该表达式断言被匹配的字符串的开头部分是 HTML 标记的开头标记。
- `\k<tag>` 后向引用名称为“tag”的分组，即被匹配的 HTML 标记的名称。
- `/` 匹配字符 `.`。
- `<\k<tag>>` 匹配 HTML 标记的结尾标记，如 ``、`</table>` 等。
- `(?=<\k<tag>>)` 是一个零宽度正预测先行断言。这里，该表达式断言被匹配的字符串的结尾部分是 HTML 标记的结尾标记。
- `.*` 匹配 HTML 标记之间的任何字符。

使用工具 Regex Tester 测试正则表达式 `(?<=<(?(tag>[a-zA-Z]\w+)\s+[>]*)).*?(?=<\k<tag>>)`，结果如图 5.28 所示。

图 5.28 测试正则表达式`(?<=<(?(tag>[a-zA-Z]\w+)\s+[^\>]*).*(?=<\k<tag>>)`

5.4.3 提取 URL

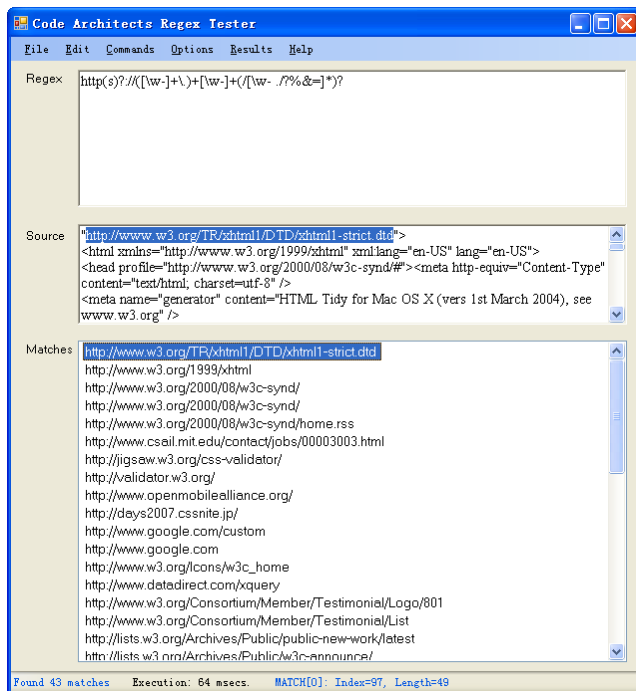
URL 一般是以字符串“http://”或“https://”开头的字符串。它可以被字符`.`、`/`、`?`、`&`、`%`或`=`分割。如 `http://www.ab.com`、`https://ac.cn`、`http://cn.net/2007/07/06/aa.aspx?ID=1` 等。以下正则表达式能够匹配 HTML 文本中的 URL，因此使用该正则表达式可以从 HTML 文本中提取 URL。

```
http(s)?://([\\w-]+\\.)+[\\w-]+(/[\\w- ./?%&=]*)? (76)
```

正则表达式解释如下。

- `http(s)?://`匹配字符串“http://”或者“https://”。
- `[\\w-]`能够匹配单词字符和连接符号`-`。
- `\\.`匹配字符`.`。
- `[\\w-]+\\.`能够匹配由单词字符和连接符号-组成的以字符串开头的、以字符`.`结尾的字符串。
- `([\\w-]+\\.)+`能够匹配 1 个或多个由单词字符和连接符号-组成的字符串开头的、以字符`.`结尾的字符串。
- `/`匹配字符`/`。
- `[\\w- ./?%&=]`能够匹配单词字符、`-`、（空格）、`.`、`/`、`?`、`%`、`&`和`=`；`[\\w- ./?%&=]*`能够匹配空字符串，或者由单词字符、`-`、（空格）、`.`、`/`、`?`、`%`、`&`和`=`组成的长度至少为 1 的字符串。
- `([\\w- ./?%&=]*)?`表示表达式`[\\w- ./?%&=]*`匹配的字符串可以不出现或者出现 1 次。

使用工具 Regex Tester 测试正则表达式 `http(s)?://([\\w-]+\\.)+[\\w-]+(/[\\w- ./?%&=]*)?`，结果如图 5.29 所示。

图 5.29 测试正则表达式 `http(s)?://([\\w-]+\\.)+[\\w-]+(/[\\w- ./?%&=]*)?`

5.4.4 提取图像的 URL

图像的 URL 包含在 `` 元素中。如果要提取图像的 URL，那么必须先匹配 `` 元素。图像的 URL 和普通 URL 一样，也是以字符串 “http://” 或 “https://” 开头的字符串。它可以被字符 `.`、`/`、`?`、`&`、`%` 或 `=` 分割。如 `http://www.ab.com`、`https://ac.cn`、`http://cn.net/2007/07/06/aa.aspx?ID=1` 等。其中，以下正则表达式能够匹配 HTML 文本中的 URL。

```
http(s)?://([\\w-]+\\.)+[\\w-]+(/[\\w- ./?%&=]*)? (77)
```

以下正则表达式能够匹配 HTML 文本中图像的 URL。

```
(?<=<img\\s*[>]*)http(s)?://([\\w-]+\\.)+[\\w-]+(/[\\w- ./?%&=]*)? (78)
```

正则表达式解释如下。

- ❑ `<img` 匹配 `` 元素的开始标记。
- ❑ `\\s*` 匹配 `` 元素的开始标记后的空白字符。
- ❑ `<img\\s*[>]*` 匹配 `` 元素中从开始标记到图像 URL 之前的内容。
- ❑ `(?<=<img\\s*[>]*)` 是一个断言，它断言被匹配的 URL 必须包含在 `` 元素中，从而保证被匹配的 URL 是图像的 URL。
- ❑ `http(s)?://` 匹配字符串 “http://” 或者 “https://”。
- ❑ `[\\w-]` 能够匹配单词字符和连接符号-。
- ❑ `\\.` 匹配字符 `.`。
- ❑ `[\\w-]+\\.` 能够匹配由单词字符和连接符号-组成的以字符串开头的、以字符 `.` 结尾的字符串。
- ❑ `([\\w-]+\\.)+` 能够匹配 1 个或多个由单词字符和连接符号-组成的、以字符串开头的、以字符 `.` 结尾的字符串。
- ❑ `/` 匹配字符 `/`。

- `[\w- ./?%&=]`能够匹配单词字符、-、（空格）、.、/、?、%、&和=；`[\w- ./?%&=]*`能够匹配空字符串，或者由单词字符、-、（空格）、.、/、?、%、&和=组成的长度至少为 1 的字符串。
- `([\w- ./?%&=]*)?`表示与表达式`[\w- ./?%&=]*`匹配的字符串可以不出现或者出现 1 次。

使用工具 Regex Tester 测试正则表达式示例（78），结果如图 5.30 所示。

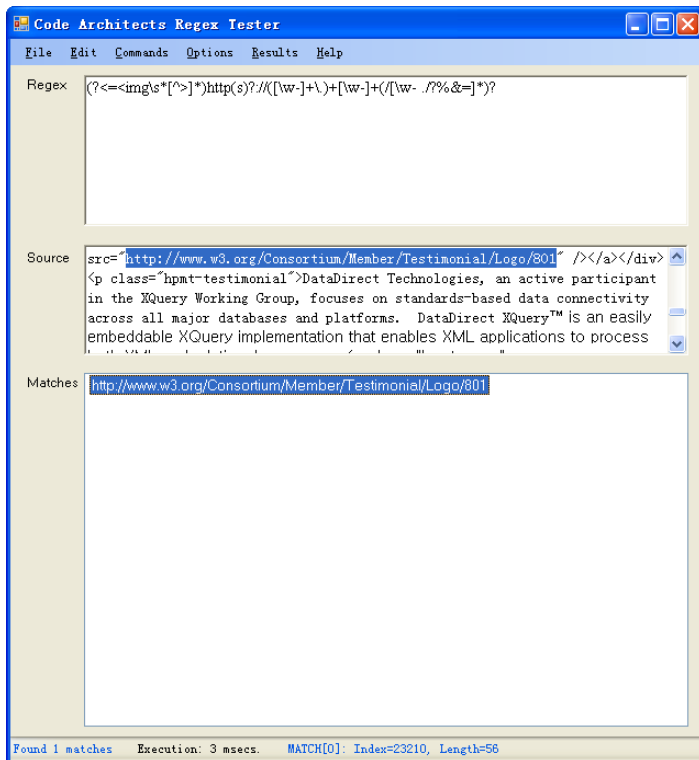


图 5.30 测试正则表达式示例（78），提取图像的 URL

5.4.5 提取 HTML 页面的标题

HTML 页面的标题放置在`<title>`元素之中，即`<title>`元素的开始标记`<title>`元素和结尾标记`</title>`元素中间的内容。以下正则表达式能够提取 HTML 页面的标题。

```
(?<=<title\s*[^>]*>).*?(?<=</title>)
```

(79)

正则表达式解释如下。

- `<title` 匹配`<title>`元素的`<title`部分。
- `\s*`匹配`<title>`元素的`<title`部分后的空白字符。
- `<title\s*[^>]*>`匹配`<title>`元素的开始标记。
- `(?<=<title\s*[^>]*>)`是一个断言，它断言被匹配的内容在`<title>`元素的开始标记之后。
- `(?<=</title>)`匹配`<title>`元素的结尾标记。
- `.*`匹配`<title>`元素的内容，即 HTML 页面的标题。

使用工具 Regex Tester 测试正则表达式“一”示例（79），结果如图 5.31 所示。

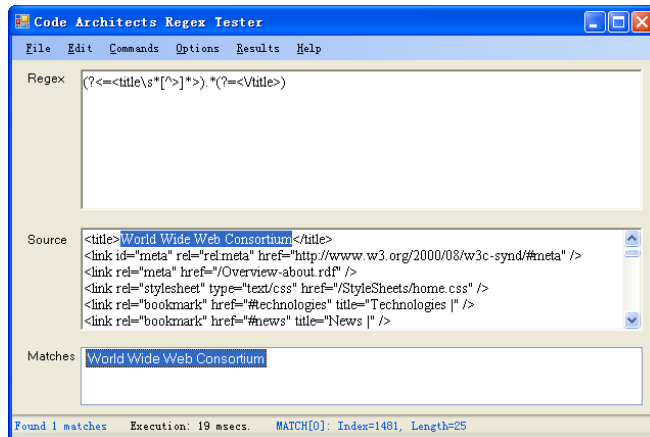


图 5.31 测试正则表达式示例 (79)，提取 HTML 页面的标题

第 2 篇 ASP.NET 正则表达式应用

第 6 章 C#常用数据类型的检查与转换

虽然用户向应用程序输入的数据或信息可以多种多样，但是这些数据或信息往往是由一个或多个字符串组成的。然而，应用程序有时只能接受如整型、浮点型、双精度型、布尔值或时间值等类型的信息。因此，应用程序常常需要把这些数据或信息转换为应用程序能够理解的数据类型，如整型、浮点型、双精度型、布尔值或时间值等。本章将介绍 C#常用数据类型的检查与转换，如数值数据类型的检查与转换、布尔数据类型的检查与转换、时间数据类型的检查与转换等。

6.1 数值数据类型的检查与转换

本节介绍数值数据类型的检查与转换功能，包括整数检查、实数检查、整数和字符串之间的转换、浮点数和字符串之间的转换。

6.1.1 整数检查

整数由数字（如果是负数，则还包括一个字符“-”）组成，包括正整数、0 和负整数。要检查一个字符串是否为整数，首先要检查该字符串是否全部由数字组成。如果不是，则该字符串不是整数。

在下面的代码示例中，函数 `CheckCharISNumber1_9(char value)` 检查一个字符是否为数字 1~9。如果是，则返回 `true`，否则返回 `false`。

```
/// <summary>
/// 检查字符是否为数字 1~9
/// </summary>
/// <param name="value"></param>
/// <returns></returns>
public static bool CheckCharISNumber1_9(char value)
{    ///检查字符是否为数字 1~9
    return value >= '1' && value <= '9';
}
```

在下面的代码示例中，函数 `CheckStringISNumber(string value)` 检查一个字符串是否全部由数字 0~9 组成。如果是，则返回 `true`，否则返回 `false`。

```
/// <summary>
/// 检查字符串是否全部由数字组成
/// </summary>
/// <param name="value"></param>
/// <returns></returns>
public static bool CheckStringISNumber(string value)
{    ///如果字符串为空，则返回 false
    if(string.IsNullOrEmpty(value) == true) return false;
    foreach(char ch in value)
    {    ///检查每一个字符串是否为数字
```



```

if(ch < '0' || ch > '9'){return false;}
}
return true;
}

```

在下面的代码示例中，函数 `CheckInt(string value)` 检查一个字符串是否为整数。如果是，则返回 `true`，否则返回 `false`。该函数的实现步骤如下。

- ❶ 判断给定字符串 `value` 是否为空。如果是，则返回 `false`，并中止函数。
- ❷ 判断给定字符串 `value` 的第一个字符是否为负号“-”。如果是，则从 `value` 中移除该字符“-”。
- ❸ 检查字符串 `value` 的长度是否为 1。如果是，则调用函数 `CheckStringISNumber(string value)` 检查该字符串是否全部由数字组成。如果是，则返回 `true`，并结束检查过程。
- ❹ 如果字符串 `value` 的长度大于 1，则调用函数 `CheckCharISNumber1_9(char value)` 检查第一个字符串是否为 1~9、函数 `CheckStringISNumber(string value)` 检查该字符串是否全部由数字组成。如果上述两个函数均返回 `true`，则被检查的字符串为整数，否则不为整数。

```

/// <summary>
/// 检查字符串是否为一个整数
/// </summary>
/// <param name="value"></param>
/// <returns></returns>
private bool CheckInt(string value)
{
    ///如果字符串为空，则返回 false
    if(string.IsNullOrEmpty(value) == true) return false;
    ///如果是负整数，则去掉前面的符号，再进行处理
    if(value[0] == '-') {value = value.Remove(0,1);}
    ///检查一位整数
    if(value.Length == 1
    && DataTypeCommonOperation.CheckStringISNumber(value))
    {
        return true;
    }
    ///检查第一个字符和整个字符串。其中，第一个字符
    return (DataTypeCommonOperation.CheckCharISNumber1_9(value[0])
    && DataTypeCommonOperation.CheckStringISNumber(value));
}

```

6.1.2 实数检查

实数由整数部分、小数部分和小数点(.)组成。其中，整数部分可以是正整数、0 和负整数。小数部分是由数字组成的字符串。要检查一个字符串是否为实数，则需要检查整数部分、小数部分，以及小数点。如果任何一部分检查失败，则字符串不为实数。

在下面的代码示例中，函数 `CheckNumber(string value)` 检查一个字符串是否为实数。如果是，则返回 `true`，否则返回 `false`。该函数的实现步骤如下。

- ❶ 判断给定字符串 `value` 是否为空。如果是，则返回 `false`，并中止函数。
- ❷ 判断给定字符串 `value` 的第一个字符是否为负号“-”。如果是，则从 `value` 中移除该字符“-”。
- ❸ 在给定字符串 `value` 中查找小数点，并使用变量 `dotIndex` 保存小数点所在的索引。如果在字符串 `value` 中未找到小数点，即变量 `dotIndex` 的值等于 -1，则调用函数 `CheckInt(string value)` 检查该字符串是否为整数，并返回检查结果。

④ 如果小数点位于第一个或最后一个位置，则该字符串 `value` 不能转换为实数。此时，返回 `false`，并中止函数。

⑤ 根据小数点的位置获取整数部分和小数部分的字符串，分别保存在变量 `intValue` 和 `decValue` 中。

⑥ 调用函数 `CheckStringISNumber(string value)` 检查小数部分，并使用变量 `isDec` 保存检查结果。如果变量 `isDec` 的值为 `false`，则返回 `false`，并中止该函数。

⑦ 分两种情况检查整数部分是否合法。其一，整数部分左边第一个字符为字符 `0`；其二，整数部分左边第一个字符不为字符 `0`。

⑧ 如果整数部分左边第一个字符不为“`0`”，则调用函数 `CheckCharISNumber1_9(char value)` 检查第一个字符串是否为 `1~9`、函数 `CheckStringISNumber(string value)` 检查该字符串是否由数字组成。

⑨ 最后，返回变量 `isDec` 和 `isInt` 的逻辑与，并结束函数。如果变量 `isDec` 和 `isInt` 的逻辑与的值为 `true`，则该字符串可以转换为一个实数，否则不能转换为一个实数。

```
/// <summary>
/// 检查字符串是否为一个实数
/// </summary>
/// <param name="value"></param>
/// <returns></returns>
private bool CheckNumber(string value)
{
    ///如果字符串为空，则返回 false
    if(string.IsNullOrEmpty(value) == true) return false;
    ///如果是负整数，则去掉前面的符号，再进行处理
    if(value[0] == '-') {value = value.Remove(0,1);}
    int dotIndex = value.IndexOf(".");
    ///如果不包含小数点，则检查是否为整数
    if(dotIndex == -1) {return CheckInt(value);}
    ///小数点位置不正确
    if(dotIndex < 1 || dotIndex > value.Length - 2) {return false;}
    ///获取小数部分和整数部分
    string intValue = value.Substring(0,dotIndex);
    string decValue = value.Substring(dotIndex + 1);
    ///检查小数部分
    bool isDec = DataTypeCommonOperation.CheckStringISNumber(decValue);
    if(isDec == false) return false;
    ///检查整数部分
    bool isInt = false;
    if(intValue[0] == '0')
    {
        if(intValue.Length == 1) isInt = true;
    }
    else
    {
        ///检查以非 0 开头的整数部分
        isInt = DataTypeCommonOperation.CheckCharISNumber1_9(intValue[0])
        && DataTypeCommonOperation.CheckStringISNumber(intValue);
    }
    return isInt && isDec;
}
```

6.1.3 整数和字符串之间的转换

把一个整数转换为字符串的方法比较简单，只要使用它的 `ToString()` 方法即可。下面的代码

示例就是把一个整数转换为其对应的字符串。

```
int thisInt = 10;
string intString = thisInt.ToString();
```

把一个字符串转换为一个整数的方法稍微复杂，可以使用 `Int32` 类的静态方法：`Parse()`和 `TryParse()`。`Parse()`方法可以把字符串转换为其相对应的整数，并将转换后的整数作为方法的返回值。`Parse()`方法存在以下 4 种重载形式。

- ❑ `int Int32.Parse(string s)`。
- ❑ `int Int32.Parse(string s, IFormatProvider provider)`。
- ❑ `int Int32.Parse(string s, NumberStyles style)`。
- ❑ `int Int32.Parse(string s, NumberStyles style, IFormatProvider provider)`。

其中，`s` 参数指定被转换的字符串；`provider` 参数提供与 `s` 参数相关的区域性特定格式设置信息；`style` 参数指定 `s` 参数允许使用的格式。

在下面的代码示例中，`ConvertToIntByParse(string value)`静态方法调用 `Parse()`方法把一个字符串转换为一个整数。如果转换运算失败，则返回-1，否则返回其相对应的整数值。另外，该方法使用 `try...catch` 语句来检查字符串能否被转换为整数。

```
/// <summary>
/// 把字符串转换为整数
/// </summary>
/// <param name="value"></param>
/// <returns></returns>
public static int ConvertToIntByParse(string value)
{
    ///如果字符串为空，则返回 false
    if(string.IsNullOrEmpty(value) == true) return -1;
    ///使用 Parse 方法转换
    int intValue = -1;
    try{intValue = Int32.Parse(value);}
    catch{}
    return intValue;
}
```

`TryParse()`方法也可以将字符串转换为其相应的整数。但是，它使用 `out` 类型的参数保存转换后的整数。`TryParse()`方法存在以下两种重载形式。

- ❑ `bool Int32.TryParse(string s, out int result)`。
- ❑ `bool Int32.TryParse(string s, NumberStyles style, IFormatProvider provider, out int result)`。

其中，`s` 参数指定被转换的字符串，`result` 参数用来保存转换后的整数，`provider` 参数提供与 `s` 参数相关的区域性特定格式设置信息，`style` 参数指定 `s` 参数允许使用的格式。

注意：`TryParse()`方法的 `result` 参数为 `out` 类型。该类型的参数常常用来保存变量在函数体中被修改的结果。

在下面的代码示例中，`ConvertToIntByTryParse(string value)`静态方法调用 `TryParse()`方法把一个字符串转换为一个整数。如果转换运算失败，则返回-1，否则返回其相对应的整数值。

```
/// <summary>
/// 把字符串转换为整数
/// </summary>
/// <param name="value"></param>
/// <returns></returns>
public static int ConvertToIntByTryParse(string value)
{
    ///如果字符串为空，则返回 false
    if(string.IsNullOrEmpty(value) == true) return -1;
```

```

///使用 TryParse 方法转换
int intValue = -1;
if(Int32.TryParse(value,out intValue) == true){return intValue;}
return -1;
}

```

6.1.4 浮点数和字符串之间的转换

把一个浮点数转换为字符串的方法比较简单，只要使用它的 ToString()方法即可。下面的代码示例就是把一个浮点数转换为其对应的字符串。

```

float thisFloat = 10.0f;
string floatString = thisFloat.ToString();

```

把一个字符串转换为一个浮点数的方法稍微复杂，可以使用 float 类的静态方法：Parse()和 TryParse()。Parse()方法可以把字符串转换为其相对应的浮点数，并将转换后的整数作为方法的返回值。Parse()方法存在以下 4 种重载形式。

注意：C#中的 float 数据类型与 .NET Framework 中的 Single 数据类型相对应。在此，仅介绍 Single 数据类型的 Parse()方法。

- ❑ float Single.Parse(string s)
- ❑ float Single.Parse(string s,IFormatProvider provider)
- ❑ float Single.Parse(string s,NumberStyles style)
- ❑ float Single.Parse(string s,NumberStyles style,IFormatProvider provider)

其中，s 参数指定被转换的字符串，provider 参数提供与 s 参数相关的区域性特定格式设置信息，style 参数指定 s 参数允许使用的格式。

在下面的代码示例中，ConvertToFloatByParse(string value)静态方法调用 Parse()方法把一个字符串转换为一个浮点数。如果转换运算失败，则返回-1，否则返回其相对应的浮点数值。另外，该方法使用 try...catch 语句来检查字符串能否被转换为浮点数。

```

/// <summary>
/// 把字符串转换为浮点数
/// </summary>
/// <param name="value"></param>
/// <returns></returns>
public static float ConvertToFloatByParse(string value)
{
    ///如果字符串为空，则返回 false
    if(string.IsNullOrEmpty(value) == true) return -1;
    ///使用 Parse 方法转换
    float floatValue = -1;
    try{floatValue = float.Parse(value);}
    catch{}
    return floatValue;
}

```

TryParse()方法也可以将字符串转换为其相应的浮点数。但是，它使用 out 类型的参数保存转换后的浮点数。TryParse()方法存在以下两种重载形式。

- ❑ bool Single.TryParse(string s,out int result)。
- ❑ bool Single.TryParse(string s, NumberStyles style, IFormatProvider provider, out int result)。

其中，s 参数指定被转换的字符串，result 参数用来保存转换后的浮点数，provider 参数提供与 s 参数相关的区域性特定格式设置信息，style 参数指定 s 参数允许使用的格式。

在下面的代码示例中，ConvertToFloatByTryParse(string value)静态方法调用 TryParse()方法把

一个字符串转换为一个浮点数。如果转换运算失败，则返回-1，否则返回其相对应的浮点数值。

```
/// <summary>
/// 把字符串转换为浮点数
/// </summary>
/// <param name="value"></param>
/// <returns></returns>
public static float ConvertToFloatByTryParse(string value)
{
    ///如果字符串为空，则返回 false
    if(string.IsNullOrEmpty(value) == true) return -1;
    ///使用 TryParse 方法转换
    float floatValue = -1;
    if(float.TryParse(value, out floatValue) == true){return floatValue;}
    return -1;
}
```

6.2 布尔数据类型检查与转换

本节介绍布尔数据类型检查与转换功能，包括布尔值检查、布尔值和字符串之间的转换。

6.2.1 布尔值检查

检查给定的字符串是否为布尔值比较简单，只要比较该字符串是否等于字符串“true”或者“false”即可。在下面的代码示例中，函数 CheckBool(string value)检查一个字符串是否为布尔值。如果是，则返回 true，否则返回 false。

```
/// <summary>
/// 检查字符串是否为布尔值
/// </summary>
/// <param name="value"></param>
/// <returns></returns>
public static bool CheckBool(string value)
{
    ///如果字符串为空，则返回 false
    if(string.IsNullOrEmpty(value) == true) return false;
    ///检查字符串是否为布尔值
    return (value.ToLower() == "true" || value.ToLower() == "false")
        ? true : false;
}
```

6.2.2 布尔值和字符串之间的转换

把一个布尔值转换为字符串的方法比较简单，只要使用它的 ToString()方法即可。下面的代码示例就是把一个布尔值转换为其对应的字符串。

```
bool thisBool = true;
string boolString = thisBool.ToString();
```

把一个字符串转换为一个布尔值的方法稍微复杂，可以使用 bool 类的静态方法：Parse()和 TryParse()。Parse()方法可以把字符串转换为其相对应的布尔值，并将转换后的布尔值作为方法的返回值。Parse()方法的原型为：bool bool.Parse(string value)。其中，value 参数指定被转换的字符串。

在下面的代码示例中，ConvertToBoolByParse(string value)静态方法调用 Parse()方法把一个字符串转换为一个布尔值。

注意：ConvertToBoolByParse(string value)静态方法返回一个可空的布尔类型（bool?）的值。如果该函数转换字符串运算失败，则返回 null；否则，返回类型为 bool?的变量 boolValue 的值，转换的结果保存在变量 boolValue 的 Value 属性中。

另外，该方法使用 try...catch 语句来检查字符串能否被转换为布尔值。

```
/// <summary>
/// 把字符串转换为布尔值
/// </summary>
/// <param name="value"></param>
/// <returns></returns>
public static bool? ConvertToBoolByParse(string value)
{    ///如果字符串为空，则返回 false
  if(string.IsNullOrEmpty(value) == true) return null;
  ///使用 Parse 方法转换
  bool? boolValue = null;
  try{boolValue = bool.Parse(value);}
  catch{}
  return boolValue;
}
```

TryParse()方法也可以将字符串转换为其相对应的布尔值。但是，它使用 out 类型的参数保存转换后的整数。TryParse()方法的原型为：bool bool.TryParse(string s,out int result)。其中，s 参数指定被转换的字符串；result 参数用来保存转换后的布尔值。

在下面的代码示例中，ConvertToBoolByTryParse(string value)静态方法调用 TryParse()方法把一个字符串转换为一个布尔值。

注意：ConvertToBoolByTryParse(string value)静态方法返回一个可空的布尔类型（bool?）的值。如果该函数转换字符串运算失败，则返回 null；否则，返回类型为 bool?的变量 boolValues 的值，转换的结果保存在变量 boolValues 的 Value 属性中。

```
/// <summary>
/// 把字符串转换为布尔值
/// </summary>
/// <param name="value"></param>
/// <returns></returns>
public static bool? ConvertToBoolByTryParse(string value)
{    ///如果字符串为空，则返回 false
  if(string.IsNullOrEmpty(value) == true) return null;
  ///使用 TryParse 方法转换
  bool boolValue;
  bool? boolValues = null;
  if(bool.TryParse(value,out boolValue) == true){boolValues = boolValue;}
  return boolValues;
}
```

6.3 时间数据类型检查与转换

本节介绍时间数据类型的检查与转换功能，包括时间值检查、时间值和字符串之间的转换。

6.3.1 时间数据类型检查

检查给定的字符串是否为时间值不是一个简单的运算。为了方便介绍后续示例，在此，设置

时间值的格式为“yyyy-MM-dd hh:mm:ss”。

在下面的代码示例中，函数 `CheckDateTime(string value)` 仅仅粗略检查一个字符串是否为时间值。如果是，则返回 `true`，否则返回 `false`。该函数的实现步骤如下。

- ① 判断给定字符串 `value` 是否为空。如果是，则返回 `false`，并中止函数。
- ② 分割给定字符串 `value`，分割后的字符串保存在数组变量 `values` 中。根据时间值的格式，`values` 变量的长度应该为 6。如果不为 6，则该字符串不是一个时间值。此时，返回 `false`，并中止函数。
- ③ 检查 `values` 变量中的每一个字符串是否全部由数字组成。如果不是，则该字符串不是一个时间值。此时，返回 `false`，并中止函数。
- ④ 如果 `values` 变量中的每一个字符串都是由数字组成，则粗略认为该字符串是一个时间值，并返回 `true`。

```
/// <summary>
/// 检查字符串是否为时间变量
/// </summary>
/// <param name="value"></param>
/// <returns></returns>
public static bool CheckDateTime(string value)
{
    ///如果字符串为空，则返回 false
    if(string.IsNullOrEmpty(value) == true) return false;
    ///分割字符串
    string[] values = value.Split(new char[] { '-', ' ', ':', '/' },
    StringSplitOptions.RemoveEmptyEntries);
    if(values.Length != 6) return false;
    foreach(string s in values)
    {
        ///检查分割后的每一个字符串是否全部数字组成
        if(DataTypeCommonOperation.CheckStringISNumber(s) == false)
        {
            return false;
        }
    }
    return true;
}
```

6.3.2 时间和字符串之间的转换

把一个时间 (`DateTime` 类型的变量) 转换为字符串的方法比较简单，只要使用它的 `ToString()` 方法即可。下面的代码示例就是把一个时间转换为其相对应的字符串。

```
int thisInt = 10;
string intString = thisInt.ToString();
```

把一个字符串转换为一个时间的方法稍微复杂，可以使用 `DateTime` 类的静态方法：`Parse()` 和 `TryParse()`。`Parse()` 方法可以把字符串转换为其相对应的时间，并将转换后的时间作为方法的返回值。`Parse()` 方法存在以下 3 种重载形式。

- ❑ `DateTime DateTime.Parse(string s)`
- ❑ `DateTime DateTime.Parse(string s, IFormatProvider provider)`
- ❑ `DateTime DateTime.Parse(string s, DateTimeStyles styles)`

其中，`s` 参数指定被转换的字符串，`provider` 参数提供与 `s` 参数相关的区域性特定格式设置信息，`styles` 参数指定 `s` 参数允许使用的格式。

在下面的代码示例中，`ConvertToDateByParse(string value)` 静态方法调用 `Parse()` 方法把一个字

字符串转换为一个时间。如果转换运算失败，则返回一个初始值（1900-1-1 00:00:00），否则返回其相对应的时间。另外，该方法使用 `try...catch` 语句来检查字符串能否被转换为时间。

```
/// <summary>
/// 把字符串转换为时间
/// </summary>
/// <param name="value"></param>
/// <returns></returns>
public static DateTime ConvertToDateByParse(string value)
{
    DateTime initValue = new DateTime(1900, 1, 1, 0, 0, 0);
    ///如果字符串为空，则返回'1900-1-1 00:00:00'
    if(string.IsNullOrEmpty(value) == true) return initValue;
    ///使用 Parse 方法转换
    try{initValue = DateTime.Parse(value);}
    catch{}
    return initValue;
}
```

`TryParse()`方法也可以将字符串转换为其相对应的时间。但是，它使用 `out` 类型的参数保存转换后的时间。`TryParse()`方法存在以下两种重载形式。

- ❑ `bool DateTime.TryParse(string s, out int result)`
- ❑ `bool DateTime.TryParse(string s, IFormatProvider provider, DateTimeStyles styles, out DateTime result)`

其中，`s` 参数指定被转换的字符串，`result` 参数用来保存转换后的时间，`provider` 参数提供与 `s` 参数相关的区域性特定格式设置信息，`styles` 参数指定 `s` 参数允许使用的格式。

在下面的代码示例中，`ConvertToDateByTryParse(string value)`静态方法调用 `TryParse()`方法把一个字符串转换为一个时间。如果转换运算失败，则返回一个初始值（1900-1-1 00:00:00），否则返回其相对应的时间。

```
/// <summary>
/// 把字符串转换为时间
/// </summary>
/// <param name="value"></param>
/// <returns></returns>
public static DateTime ConvertToDateByTryParse(string value)
{
    DateTime initValue = new DateTime(1900, 1, 1, 0, 0, 0);
    ///如果字符串为空，则返回'1900-1-1 00:00:00'
    if(string.IsNullOrEmpty(value) == true) return initValue;
    ///使用 TryParse 方法转换
    DateTime.TryParse(value, out initValue);
    return initValue;
}
```

6.4 数据类型检查与转换应用实例

为了测试数据类型检查功能，创建了 ASP.NET 应用程序 `DataTypeApplication`。该应用程序在 `App_Code/DataTypeCommonOperation.cs` 文件中声明了名字空间 `RegexExpression`，其程序代码如下：

```
namespace RegexExpression
{
    ///.....
}
```


下面的代码示例为 ASP.NET 应用程序 DataTypeApplication 中的 DataTypeCheck.aspx.cs 文件的部分内容。该文件首先引入了自定义的名字空间 RegexpExpression, 并且在 Page_Load(object sender, EventArgs e) 事件中测试了各种数据类型检查的函数。

```
///引入新的名字空间
using RegexpExpression;
public partial class DataTypeCheck : System.Web.UI.Page
{
protected void Page_Load(object sender, EventArgs e)
{
Response.Write("检查\"123456789\"是否为整数。检查结果为: "
+ CheckInt("123456789") + "<br />");
Response.Write("检查\"aa0123456789bb\"是否为整数。检查结果为: "
+ CheckInt("aa0123456789bb") + "<br />");
Response.Write("检查\"-0.1234567890000\"是否为实数。检查结果为: "
+ CheckNumber("-0.1234567890000") + "<br />");
Response.Write("检查\"-01234f.1234567890000\"是否为实数。检查结果为: "
+ CheckNumber("-01234f.1234567890000") + "<br />");
Response.Write("检查\"true\"是否为布尔值。检查结果为: "
+ CheckBool("true") + "<br />");
Response.Write("检查\"class\"是否为布尔值。检查结果为: "
+ CheckBool("class") + "<br />");
Response.Write("检查\"2007-07-27 08:00:00\"是否为时间值。检查结果为: "
+ CheckDateTime("2007-07-27 08:00:00") + "<br />");
Response.Write("检查\"2007-07/27 08:00:00 am\"是否为时间值。检查结果
为: " + CheckDateTime("2007-07/27 08:00:00 am") + "<br />");
}
}
///以下代码省略, 读者可以参考 DataTypeApplication/DataTypeCheck.aspx.cs 文件
}
```

把 DataTypeCheck.aspx 页面设置为起始页面, 并运行 ASP.NET 应用程序 DataTypeApplication, 结果如图 6.1 所示。该页面显示了各个函数的测试结果。

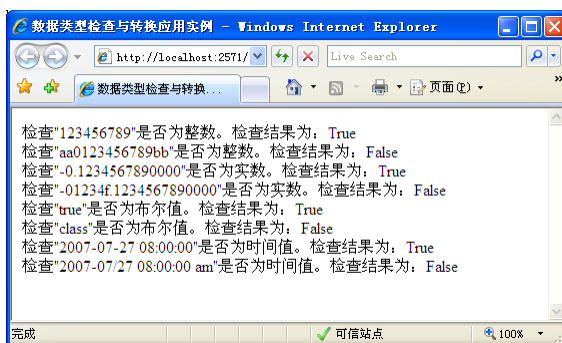


图 6.1 DataTypeCheck.aspx 页面显示了各个函数的测试结果

第 7 章 不可变字符串与可变字符串的处理

一般情况下，字符串是程序中最常用的数据类型，几乎所有的程序语言都提供了处理字符串的方法或函数。当然，.NET Framework 也提供了丰富的处理字符串的方法或函数。

.NET Framework 把字符串分为不可变字符串（由 `String` 类表示）和可变字符串（由 `StringBuilder` 类表示）。其中，不可变字符串对象一旦被创建，那么该对象是不能被修改的；而可变字符串对象被创建之后，开发人员可以对该对象进行修改，如追加、移除、替换、插入等运算。

本章将介绍不可变字符串（由 `String` 类表示）与可变字符串（由 `StringBuilder` 类表示）的处理方法。

7.1 15 种不可变字符串 `String` 处理

`String` 类表示不可变字符串，它能够用来表示文本。这些文本一般是由 Unicode 字符组成的字符串。本节将介绍不可变字符串的处理方法。

7.1.1 `String` 类和对象

`String` 类可以用来表示文本，即由 Unicode 字符组成的字符串。`String` 对象是 `Char` 对象的有序集合。

注意：`String` 是引用类型，但是它的内容是不可改变的。即一个 `String` 对象一旦被创建，那么该对象的内容是不能被修改的。

下面的代码示例声明了两个字符串变量 `strA` 和 `strB`，并且它们的值分别为“abcd”和“1234”。此时，系统的内存分配如图 7.1 所示。

```
string strA = "abcd";  
string strB = "1234";
```

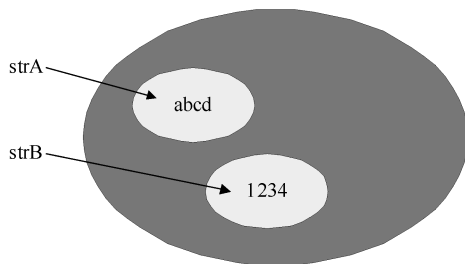


图 7.1 变量 `strA` 和 `strB` 的内存分配图

下面的代码示例修改了字符串变量 `strA` 的值，并设置它的值为变量 `strA` 在修改前的值与变量 `strB` 的值之和（在此，和表示字符串拼接）。因此，变量 `strA` 和变量 `strB` 的值分别为“abcd1234”和“1234”。此时，系统的内存分配如图 7.2 所示。

```
string strA = strA + strB;
```

注意：在上述程序代码中，变量 strA 被修改之后，系统实际上为其重新分配了一块新内存来保存字符串“abcd1234”。变量 strA 被修改之前分配的内存已经不能被访问了。因此，如果代码中包含大量的字符串赋值运算，那么程序将至少为每一个字符串运算都重新分配一块新内存。这类代码是非常消耗系统资源的。

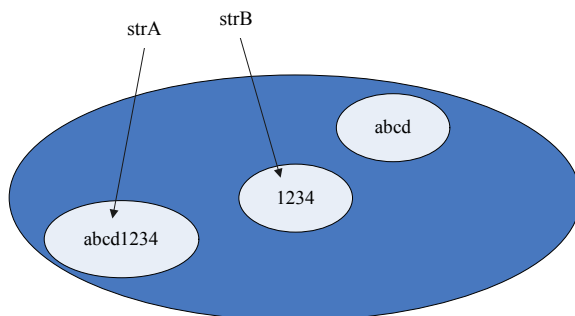


图 7.2 修改运算之后的变量 strA 和 strB 的内存分配图

String 类提供了 1 个公共字段和 2 个公共属性，它们的具体说明如表 7.1 所示。

表 7.1 String 类的字段和属性

字段和属性	说明
Empty（静态字段）	空字符串
Length	实例中的字符数，即字符串的长度
Chars	获取实例中指定字符位置的字符

7.1.2 插入字符串

Insert()方法可以向字符串（String 类的实例）中的指定位置插入一个字符串。它的原形如下：
`public string Insert(int startIndex,string value)。`

其中，startIndex 参数指定插入的索引位置，value 参数指定被插入的字符串。

下面的示例代码使用 Insert()方法向 String 类的实例 initValue 插入了字符串“New string.---”。另外，InsertString()函数还使用 Response.Write()方法分别在网页上显示了插入运算之前和之后的字符串。最后，InsertString()函数返回变量 initValue 的值。

```
private string InsertString()
{
    string initValue = "This is a string.";
    Response.Write("插入运算前的字符串: " + initValue + "<br />");
    ///执行插入运算
    string newValue = initValue.Insert(0,"New string.---");
    Response.Write("插入运算后的字符串: " + newValue + "<br />");
    return newValue;
}
```

7.1.3 替换字符串

Replace()方法可以将字符串（String 类的实例）中指定的字符或字符串替换为其他的字符或字符串。它有两种重载形式，具体如下。

- ❑ `public string Replace(char oldChar,char newChar)`，将指定的字符替换为其他的字符。
- ❑ `public string Replace(string oldValue,string newValue)`，将指定的字符串替换为其他的字符串。

其中，oldChar 参数指定被替换的字符，newChar 参数为替换的字符，oldValue 参数指定被替换的字符串，newValue 参数为替换的字符串。

下面的示例代码使用 Replace()方法把 String 类的实例 initValue 中的字符串“string”替换为“STRING”。另外，ReplaceString()函数还使用了 Response.Write()方法，分别在网页上显示了替换运算之前和之后的字符串。最后，ReplaceString()函数返回替换运算之后的字符串变量 newValue 的值。

```
private string ReplaceString()
{
    string initValue = "This is a string.";
    Response.Write("替换运算前的字符串: " + initValue + "<br />");
    ///执行替换运算
    string newValue = initValue.Replace("string", "STRING");
    Response.Write("替换运算后的字符串: " + newValue + "<br />");
    return newValue;
}
```

7.1.4 填充字符串

PadLeft()和 PadRight()方法均能够填充字符串。它们都可以用空白字符或指定的字符填充字符串，以达到指定的长度。其中，PadLeft()方法在字符串的左边填充，PadRight()方法在字符串的右边填充。PadLeft()和 PadRight()方法的重载形式如下。

- ❑ public string PadLeft(int totalWidth): 在字符串左边填充空白字符串，以达到指定的长度（由 totalWidth 参数指定）。
- ❑ public string PadLeft(int totalWidth, char paddingChar): 在字符串左边填充指定的字符（由 paddingChar 参数指定），以达到指定的长度（由 totalWidth 参数指定）。
- ❑ public string PadRight(int totalWidth): 在字符串右边填充空白字符串，以达到指定的长度（由 totalWidth 参数指定）。
- ❑ public string PadRight(int totalWidth, char paddingChar): 在字符串右边填充指定的字符（由 paddingChar 参数指定），以达到指定的长度（由 totalWidth 参数指定）。

下面的示例代码分别使用了 PadLeft()和 PadRight()方法填充了 String 类的实例 initValue。这两个方法的填充方式如下。

- ❑ PadLeft()方法是在 String 类的实例 initValue 中填充长度为 10 的字符串。如果填充前，实例 initValue 的长度小于 10，则在实例 initValue 的左边填补字符“A”，直到填充后实例 initValue 的长度为 10。
- ❑ PadRight()方法是在 String 类的实例 initValue 中填充长度为 10 的字符串。如果填充前，实例 initValue 的长度小于 10，则在实例 initValue 的右边填补字符“B”，直到填充后实例 initValue 的长度为 10。

另外，PadString()函数还使用了 Response.Write()方法，首先在网页上显示了填充运算之前的字符串，然后分别显示了左填充和右填充之后的字符串。最后，PadString()函数返回右填充运算之后的字符串变量 newValue 的值。

```
private string PadString()
{
    string initValue = "This is a string.";
    Response.Write("填充运算前的字符串: " + initValue + "<br />");
    ///执行左边填充运算
```

```
string newValue = initValue.PadLeft(initValue.Length + 10, 'A');
Response.Write("左边填充运算后的字符串: " + newValue + "<br />");
///执行右边填充运算
newValue = initValue.PadRight(initValue.Length + 10, 'B');
Response.Write("右边填充运算后的字符串: " + newValue + "<br />");
return newValue;
}
```

7.1.5 删除字符串

Remove()方法可以从字符串（String 类的实例）中删除指定的字符或字符串。它有两种重载形式，具体说明如下。

- ❑ public string Remove(int startIndex), 删除从指定索引位置开始的所有字符。
- ❑ public string Remove(int startIndex,int count), 删除从指定索引位置开始的指定数量字符。

其中，startIndex 参数指定被删除字符的开始索引位置，count 参数指定被删除的字符数量。

下面的示例代码使用 Remove()方法从 String 类的实例 initValue 中删除了前一半的字符。另外，RemoveString()函数还使用 Response.Write()方法分别在网页上显示删除运算之前和之后的字符串。最后，RemoveString()函数返回删除运算之后的字符串变量 newValue 的值。

```
private string RemoveString()
{
    string initValue = "This is a string.";
    Response.Write("删除运算前的字符串: " + initValue + "<br />");
    ///执行删除运算
    string newValue = initValue.Remove(0, initValue.Length / 2);
    Response.Write("删除运算后的字符串: " + newValue + "<br />");
    return newValue;
}
```

7.1.6 分割字符串

Split()方法能够将给定的字符串分割为多个子字符串，并返回由子字符串组成的字符串数组。它有 5 种重载形式，具体如下。

- ❑ public string[] Split(params char[] separator)。
- ❑ public string[] Split(char[] separator,int count)。
- ❑ public string[] Split(char[] separator,StringSplitOptions options)。
- ❑ public string[] Split(string[] separator,StringSplitOptions options)。
- ❑ public string[] Split(char[] separator,int count,StringSplitOptions options)。
- ❑ public string[] Split(string[] separator,int count,StringSplitOptions options)。

其中，separator 参数为分割字符串的字符数组或字符串数组；count 参数指定返回的子字符串的最大数量；options 参数指定字符串分割选项，它的值可以是 StringSplitOptions.None 或 StringSplitOptions.RemoveEmptyEntries，其中，第一个选项表示返回包括空字符串的数组，第二个选项表示返回不包括空字符串的数组。

下面的代码示例使用了 Split()方法对 String 类的实例 initValue 进行分割。其中，分割字符为“ ”（空白字符）。实例 initValue 分割之后，返回一个字符串数组，并保存在变量 newValue 中。另外，SplitString()函数还使用了 Response.Write()方法在网页上显示了分割运算之前的字符串，以及分割后的每一个字符串。

```
private string SplitString()
```

```

{
string initValue = "This is a string.";
Response.Write("分割运算前的字符串: " + initValue + "<br />");
///执行分割运算
string[] newValue = initValue.Split(new char[] { ' ' },
StringSplitOptions.RemoveEmptyEntries);
Response.Write("分割运算后的字符串: ");
for(int i = 0; i < newValue.Length; i++)
{
Response.Write(newValue[i] + "<br />");
}
Response.Write("<br />");
return initValue;
}

```

7.1.7 比较字符串

比较两个字符串,可以使用 `Compare()`、`CompareOrdinal()`和 `CompareTo()`方法。其中,`Compare()`方法比较两个指定的字符串,`CompareOrdinal()`方法通过计算两个字符串中相应字符的数值来比较这两个字符串,`CompareTo()`方法将当前的字符串与其他字符串进行比较。

`Compare()`方法有 8 种重载形式,具体如下。

- ❑ `public static int Compare(string strA,string strB)`
- ❑ `public static int Compare(string strA,string strB,bool ignoreCase)`
- ❑ `public static int Compare(string strA,string strB,StringComparison comparisonType)`
- ❑ `public static int Compare(string strA,string strB,bool ignoreCase,CultureInfo culture)`
- ❑ `public static int Compare(string strA,int indexA,string strB,int indexB,int length)`
- ❑ `public static int Compare(string strA,int indexA,string strB,int indexB,int length,bool ignoreCase)`
- ❑ `public static int Compare(string strA,int indexA,string strB,int indexB,int length,StringComparison comparisonType)`
- ❑ `public static int Compare(string strA,int indexA,string strB,int indexB,int length,bool ignoreCase,CultureInfo culture)`

其中, `strA` 和 `strB` 参数指定比较的字符串, `ignoreCase` 参数指定是否要忽略大小写, `comparisonType` 参数指定比较时所使用的区域、大小写和排序规则, `culture` 参数提供特定区域性的比较设置信息, `indexA` 和 `indexB` 参数分别指定 `strA` 和 `strB` 内子字符串的位置, `length` 参数指定比较的子字符串中字符的最大数量。

`CompareOrdinal()`方法有两种重载形式,具体如下。

- ❑ `public static int CompareOrdinal(string strA, string strB)`
- ❑ `public static int CompareOrdinal(string strA,int indexA,string strB,int indexB,int length)`

其中, `strA` 和 `strB` 参数指定比较的字符串, `indexA` 和 `indexB` 参数分别指定 `strA` 和 `strB` 内子字符串的位置, `length` 参数指定比较的子字符串中字符的最大数量。

`CompareTo()`方法有两种重载形式,具体如下。

- ❑ `public int CompareTo(Object value);`
- ❑ `public int CompareTo(string strB);`

其中, `value` 参数表示被比较的对象, `strB` 参数表示被比较的字符串。

注意：在上述比较方法中，如果字符串 strA 大于字符串 strB，则方法返回大于 0 的整数；如果字符串 strA 等于字符串 strB，则方法返回 0；如果字符串 strA 小于字符串 strB，则方法返回小于 0 的整数。

下面的代码示例介绍了字符串比较的方法。Compare()方法比较了字符串“one”和字符串“ONE”；CompareOrdinal()方法比较了字符串“one”和字符串“ONE”；String 类的实例 leftValue 使用 CompareTo()方法与字符串“jack”进行比较。另外，CompareString()函数还使用了 Response.Write()方法在网页上显示了上述比较运算的结果。

```
private string CompareString()
{
    string leftValue = "tom";
    string rightValue = "jack";
    ///执行比较运算
    Response.Write("string.Compare(\"one\", \"ONE\") 的结果为: "
    + string.Compare("one", "ONE") + "<br />");
    Response.Write("string.CompareOrdinal(\"one\", \"ONE\") 的结果为: "
    + string.CompareOrdinal("one", "ONE") + "<br />");
    ///执行比较运算
    Response.Write("\"tom\" 比较 \"jack\" 的结果为: "
    + leftValue.CompareTo(rightValue).ToString() + "<br />");
    return leftValue + rightValue;
}
```

7.1.8 连接字符串

Concat()方法能够将一个或多个字符串，或者一个或多个对象的字符串表示形式连接起来，从而构成一个新的字符串。Join()方法能够使用分割符号将指定字符串数组的每个元素串连起来，从而构成一个新的字符串。

Concat()方法有 9 种重载形式，具体如下。

- ❑ public static string Concat(Object arg0)
- ❑ public static string Concat(params Object[] args)
- ❑ public static string Concat(params string[] values)
- ❑ public static string Concat(Object arg0, Object arg1)
- ❑ public static string Concat(string str0, string str1)
- ❑ public static string Concat(Object arg0, Object arg1, Object arg2)
- ❑ public static string Concat(string str0, string str1, string str2)
- ❑ public static string Concat(Object arg0, Object arg1, Object arg2, Object arg3)
- ❑ public static string Concat(string str0, string str1, string str2, string str3)

其中，arg0、arg1、arg2 和 arg3 参数表示被连接的对象，str0、str1、str2 和 str3 参数表示被连接的字符串，args 参数存放被连接的对象所在的数组，values 参数存放被连接的字符串所在的数组。

Join()方法有两种重载形式，具体如下。

- ❑ public static string Join(string separator, string[] value)
- ❑ public static string Join(string separator, string[] value, int startIndex, int count)

其中，separator 参数指定各个字符串之间的分割字符，value 参数指定被连接的字符串所在的数组，startIndex 参数表示连接时使用的 value 参数中的第一个字符串，count 参数指定使用的 value

参数中的元素数。

下面的代码使用了 `Concat()` 方法和 `Join()` 方法来连接字符串。其中，`Concat()` 方法连接字符串 “This is a string.”（即 `String` 类的实例 `initValue`）和字符串 “---New string.”；`Join()` 方法使用连接字符 “,” 把字符串数组 `values` 中的每一个字符串都连接起来，并构成一个新的字符串。另外，`JoinString()` 函数还使用了 `Response.Write()` 方法在网页上显示了连接运算之前的字符串，以及使用 `Concat()` 和 `Join()` 方法连接之后的字符串。

```
private string JoinString()
{
    string initValue = "This is a string.";
    Response.Write("连接运算前的字符串: " + initValue + "<br />");
    ///执行连接运算
    string newValue = string.Concat(initValue, "---New string.");
    Response.Write("连接运算后的字符串: " + newValue + "<br />");
    ///执行 Join 运算
    string[] values = new string[10];
    for(int i = 0; i < values.Length; i++)
    {
        values[i] = i.ToString();
    }
    newValue = string.Join(",", values);
    Response.Write("Join 运算后的字符串: " + newValue + "<br />");
    return newValue;
}
```

7.1.9 处理字符串中的空白

`Trim()` 方法能够从字符串的开始和末尾移除指定的字符。`TrimStart()` 方法能够从字符串的开始位置移除与指定数组中相同的字符。`TrimEnd()` 方法能够从字符串的结尾位置移除与指定数组中相同的字符。`Trim()`、`TrimStart()` 和 `TrimEnd()` 方法的重载形式如下。

- ❑ `public string Trim()`
- ❑ `public string Trim(params char[] trimChars)`
- ❑ `public string TrimEnd(params char[] trimChars)`
- ❑ `public string TrimStart(params char[] trimChars)`

其中，`Trim()` 方法能够移除字符串的开始位置和末尾位置的空白字符，`trimChars` 参数指定被移除的字符组成的数组。

下面的代码能够使用了 `Trim()` 和 `TrimStart()` 方法处理 `String` 类的实例 `initValue` 中的空白字符。其中，`Trim()` 方法移除实例 `initValue` 开头和结尾处的空白字符，`TrimStart()` 方法仅仅去掉实例 `initValue` 开头处的空白字符。另外，`BlankString()` 函数还使用了 `Response.Write()` 方法首先在网页上显示了实例 `initValue` 的内容，然后显示了处理空白字符之后的字符串。

注意：为了能够显示空白字符，`BlankString()` 函数特意对显示字符串进行了 URL 编码。因此，空白字符会显示为字符串 “%20”。

```
private string BlankString()
{
    string initValue = "  This is a string.  ";
    Response.Write("未处理之前的字符串（已编码）: "
        + Server.UrlEncode(initValue) + "<br />");
    ///移除两端空白
```



```
string newValue = initValue.Trim();
Response.Write("移除两端的空白后的字符串（已编码）： "
+ Server.UrlEncode(newValue) + "<br />");
///移除开始处的空白
newValue = initValue.TrimStart(new char[]{' '});
Response.Write("移除开始处的空白后的字符串（已编码）： "
+ Server.UrlEncode(newValue) + "<br />");
return newValue;
}
```

7.1.10 转换字符串大小写

ToUpper()和 ToLower()方法分别把字符串转换为其大写和小写形式，它们的重载形式如下。

- ❑ public string ToUpper()
- ❑ public string ToUpper(CultureInfo culture)
- ❑ public string ToLower()
- ❑ public string ToLower(CultureInfo culture)

其中，culture 参数提供区域性特定的大小写规则。

注意：除了上述两种方法之外，String 类还提供了 ToUpperInvariant()和 ToLowerInvariant()方法。这两种方法使用当前运算系统所在区域的大小写规则，分别把给定的字符串转换为大写形式和小写形式。特别地，如果要获取与运算系统相关的信息，那么可以使用这两种方法。

下面的代码使用了 ToLower()和 ToUpper()方法把 String 类的实例 initValue 分别转换为全部由小写字母组成的字符串和全部由大写字母组成的字符串。另外，UpperLitterString()函数还使用了 Response.Write()方法在网页上显示了大小写转换运算之前的字符串，以及大写转换和小写转换后的字符串。

```
private string UpperLitterString()
{
    string initValue = "This is A StrIng.";
    Response.Write("未处理之前的字符串： " + initValue + "<br />");
    ///执行小写转换运算
    string newValue = initValue.ToLower();
    Response.Write("小写转换后的字符串： " + newValue + "<br />");
    ///执行大写转换运算
    newValue = initValue.ToUpper();
    Response.Write("大写转换后的字符串： " + newValue + "<br />");
    return newValue;
}
```

7.1.11 匹配和检索字符串

Contains()方法能够检查给定的字符串中是否包含其他的字符串。如果存在，则返回 true，否则返回 false。它的原形如下。

```
public bool Contains(string value);
```

其中，value 参数指定被检查的字符串。

EndsWith()和 StartsWith()方法能够分别判断字符串是否以指定的字符串开头或结束。如果是，则返回 true，否则返回 false。它们的重载形式如下。

- ❑ `public bool EndsWith(string value)`
- ❑ `public bool EndsWith(string value,StringComparison comparisonType)`
- ❑ `public bool EndsWith(string value,bool ignoreCase,CultureInfo culture)`
- ❑ `public bool StartsWith(string value)`
- ❑ `public bool StartsWith(string value,StringComparison comparisonType)`
- ❑ `public bool StartsWith(string value,bool ignoreCase,CultureInfo culture)`

其中，`value` 参数为被比较的字符串。`comparisonType` 参数指定字符串与 `value` 参数的比较方式，`ignoreCase` 参数指定是否忽略大小写，`culture` 参数提供比较运算的区域性信息。

`IndexOf()`和 `LastIndexOf()`方法能够在字符串中查找给定的字符串。如果存在，则返回第一个匹配字符的索引；如果不存在，则返回-1。其中，`IndexOf()`方法从字符串的开始处查找给定的字符串；`LastIndexOf()`方法从字符串的末尾处查找给定的字符串。

- ❑ `public int IndexOf(char value)`: 查找是否存在指定的字符（由 `value` 参数指定）。
- ❑ `public int IndexOf(string value)`: 查找是否存在指定的字符串（由 `value` 参数指定）。
- ❑ `public int IndexOf(char value,int startIndex)`: 查找是否存在指定的字符（由 `value` 参数指定），并指定查找运算的起始位置（由 `startIndex` 参数指定）。
- ❑ `public int IndexOf(string value,int startIndex)`: 查找是否存在指定的字符串（由 `value` 参数指定），并指定查找运算的起始位置（由 `startIndex` 参数指定）。
- ❑ `public int IndexOf(string value,StringComparison comparisonType)`: 查找是否存在指定的字符串（由 `value` 参数指定），并指定查找运算所使用的区域、大小写和排序规则（由 `comparisonType` 参数指定）。
- ❑ `public int IndexOf(char value,int startIndex,int count)`: 查找是否存在指定的字符（由 `value` 参数指定），并指定查找运算的起始位置（由 `startIndex` 参数指定）和被查找的字符数量（由 `count` 参数指定）。
- ❑ `public int IndexOf(string value,int startIndex,int count)`: 查找是否存在指定的字符串（由 `value` 参数指定），并指定查找运算的起始位置（由 `startIndex` 参数指定）和被查找的字符数量（由 `count` 参数指定）。
- ❑ `public int IndexOf(string value,int startIndex,StringComparison comparisonType)`: 查找是否存在指定的字符串（由 `value` 参数指定），并指定查找运算的起始位置（由 `startIndex` 参数指定），以及所使用的区域、大小写和排序规则（由 `comparisonType` 参数指定）。
- ❑ `public int IndexOf(string value,int startIndex,int count,StringComparison comparisonType)`: 查找是否存在指定的字符串（由 `value` 参数指定），并指定查找运算的起始位置（由 `startIndex` 参数指定）、被查找的字符数量（由 `count` 参数指定），以及所使用的区域、大小写和排序规则（由 `comparisonType` 参数指定）。
- ❑ `public int LastIndexOf(char value)`: 查找是否存在指定的字符（由 `value` 参数指定）。
- ❑ `public int LastIndexOf(string value)`: 查找是否存在指定的字符串（由 `value` 参数指定）。
- ❑ `public int LastIndexOf(char value,int startIndex)`: 查找是否存在指定的字符（由 `value` 参数指定），并指定查找运算的起始位置（由 `startIndex` 参数指定）。
- ❑ `public int LastIndexOf(string value,int startIndex)`: 查找是否存在指定的字符串（由 `value` 参数指定），并指定查找运算的起始位置（由 `startIndex` 参数指定）。
- ❑ `public int LastIndexOf(string value,StringComparison comparisonType)`: 查找是否存在指定

的字符串（由 `value` 参数指定），并指定查找运算所使用的区域、大小写和排序规则（由 `comparisonType` 参数指定）。

- ❑ `public int LastIndexOf(char value,int startIndex,int count)`: 查找是否存在指定的字符（由 `value` 参数指定），并指定查找运算的起始位置（由 `startIndex` 参数指定）和被查找的字符数量（由 `count` 参数指定）。
- ❑ `public int LastIndexOf(string value,int startIndex,int count)`: 查找是否存在指定的字符串（由 `value` 参数指定），并指定查找运算的起始位置（由 `startIndex` 参数指定）和被查找的字符数量（由 `count` 参数指定）。
- ❑ `public int LastIndexOf(string value,int startIndex,StringComparison comparisonType)`: 查找是否存在指定的字符串（由 `value` 参数指定），并指定查找运算的起始位置（由 `startIndex` 参数指定），以及所使用的区域、大小写和排序规则（由 `comparisonType` 参数指定）。
- ❑ `public int LastIndexOf(string value,int startIndex,int count,StringComparison comparisonType)`: 查找是否存在指定的字符串（由 `value` 参数指定），并指定查找运算的起始位置（由 `startIndex` 参数指定）、被查找的字符数量（由 `count` 参数指定），以及所使用的区域、大小写和排序规则（由 `comparisonType` 参数指定）。

`IndexOfAny()`和 `LastIndexOfAny()`方法能够从字符串中查找给定的字符数组中匹配项的索引。其中，`IndexOfAny()`方法返回第一个匹配项的索引，`LastIndexOfAny()`方法返回最后一个匹配项的索引。它们的重载形式如下。

- ❑ `public int IndexOfAny(char[] anyOf)`
- ❑ `public int IndexOfAny(char[] anyOf,int startIndex)`
- ❑ `public int IndexOfAny(char[] anyOf,int startIndex,int count)`
- ❑ `public int LastIndexOfAny(char[] anyOf)`
- ❑ `public int LastIndexOfAny(char[] anyOf,int startIndex)`
- ❑ `public int LastIndexOfAny(char[] anyOf,int startIndex,int count)`

其中，`anyOf` 参数指定被比较的字符数组，`startIndex` 参数指定字符位置开始处，`count` 参数指定比较的字符数量。

下面的代码使用 `Contains()`方法检查字符串变量 `initValue` 是否包含字符串“string”；使用 `StartsWith()`方法检查字符串变量 `initValue` 是否以字符串“This”开头；使用 `IndexOf()`方法查找字符串“is a s”在字符串变量 `initValue` 中的位置，如果找到，则返回第一个匹配项的开始索引，否则返回-1。`MatchString()`函数首先使用 `Response.Write()`方法输出字符串变量 `initValue` 的值，然后分别输出上述 3 个检查或查找运算的结果。

```
private string MatchString()
{
    string initValue = "This is a string.";
    Response.Write("源字符串: " + initValue + "<br />");
    ///检查是否包含
    Response.Write("源字符串是否包含字符串\"string\": "
    + initValue.Contains("string").ToString() + "<br />");
    ///检查开头
    Response.Write("源字符串是否以字符串\"This\": "
    + initValue.StartsWith("This").ToString() + "<br />");
    ///检索字符串
    Response.Write("源字符串是否存在字符串\"is a s\": "
    + (initValue.IndexOf("is a s") > -1 ? true : false).ToString())
}
```

```
+ "<br />");
return initValue;
}
```

7.1.12 格式化字符串

Format()方法能够格式化指定的字符串，即能够将指定的字符串中的每个格式项替换为相应对象的值的文本等效项。该方法有 5 种重载形式，具体如下。

- ❑ public static string Format(string format, Object arg0)
- ❑ public static string Format(string format, params Object[] args)
- ❑ public static string Format(IFormatProvider provider, string format, params Object[] args)
- ❑ public static string Format(string format, Object arg0, Object arg1)
- ❑ public static string Format(string format, Object arg0, Object arg1, Object arg2)

其中，format 参数指定格式化项，arg0、arg1、arg2 和 arg3 参数表示被格式化的对象，args 参数为被格式化对象所在的数组，provider 参数提供区域性特定的格式设置信息。

下面的代码使用了 Format()方法格式化 String 类的实例 initValue，并在该实例之前添加了字符串“这是格式化后的字符串:”。其中，格式化后的字符串保存在变量 newValue 中。另外，FormatString()函数还使用 Response.Write()方法分别在网页上显示了源字符串和格式化后的字符串。最后，FormatString()函数返回格式化后的字符串变量 newValue 的值。

```
private string FormatString()
{
    string initValue = "This is a string.";
    Response.Write("源字符串: " + initValue + "<br />");
    ///执行插入运算
    string newValue = string.Format("这是格式化后的字符串:{0}",initValue);
    Response.Write("格式化后的字符串: " + newValue + "<br />");
    return newValue;
}
```

7.1.13 获取子字符串

Substring()方法能够从给定的字符串中检索符合条件的子字符串。该方法有两种重载形式，具体如下。

- ❑ public string Substring(int startIndex)
- ❑ public string Substring(int startIndex, int length)

其中，startIndex 参数指定检索子字符串开始的位置，length 参数指定子字符串的长度。

下面的代码示例使用了 Substring()方法获取了 String 类的实例 initValue 中的子字符串。其中，子字符串为实例 initValue 的前半部分，并保存在变量 newValue 中。SubString()函数使用 Response.Write()方法分别在网页上显示了源字符串（即实例 initValue）和子字符串。最后，SubString()函数返回子字符串 newValue 的值。

```
private string SubString()
{
    string initValue = "This is a string.";
    Response.Write("源字符串: " + initValue + "<br />");
    ///获取子字符串
    string newValue = initValue.Substring(0,initValue.Length / 2);
}
```

```
Response.Write("子字符串: " + newValue + "<br />");
return newValue;
}
```

7.1.14 编码字符串

在下面的代码中，EncodingString()函数对 String 类的实例 initValue 进行了 4 种编码：ASCII、UTF7、UTF8 和 Unicode。EncodingString()函数在对实例 initValue 编码之后，输出每一编码之后的字符串。最后，EncodingString()函数返回实例 initValue 的值。

```
private string EncodingString()
{
    string initValue = "This is a string.字符串.";
    Response.Write("源编码前的字符串: " + initValue + "<br />");
    ///执行 ASCII 编码
    byte[] bvalues = Encoding.ASCII.GetBytes(initValue);
    Response.Write("ASCII 编码后的数组值: ");
    for(int i = 0; i < bvalues.Length; i++)
    {
        Response.Write(bvalues[i] + ",");
    }
    Response.Write("<br />");
    ///执行 UTF7 编码
    bvalues = Encoding.UTF7.GetBytes(initValue);
    Response.Write("UTF7 编码后的数组值: ");
    for(int i = 0; i < bvalues.Length; i++)
    {
        Response.Write(bvalues[i] + ",");
    }
    Response.Write("<br />");
    ///执行 UTF8 编码
    bvalues = Encoding.UTF8.GetBytes(initValue);
    Response.Write("UTF8 编码后的数组值: ");
    for(int i = 0; i < bvalues.Length; i++)
    {
        Response.Write(bvalues[i] + ",");
    }
    Response.Write("<br />");
    ///执行 Unicode 编码
    bvalues = Encoding.Unicode.GetBytes(initValue);
    Response.Write("Unicode 编码后的数组值: ");
    for(int i = 0; i < bvalues.Length; i++)
    {
        Response.Write(bvalues[i] + ",");
    }
    Response.Write("<br />");
    return initValue;
}
```

7.1.15 不可变字符串 String 处理的应用

ASP.NET 应用程序 StringApplication 演示了不可变字符串 String 的处理方法。该应用程序添加了名称为 StringDealwith.aspx 的页面，并测试了本节中的插入、删除、替换、分割和填充字符串等运算，以及编码和格式化字符串的方法。

下面的代码为 StringDealwith.aspx.cs 文件的部分内容。该文件在 Page_Load(object sender,

EventArgs e)事件中测试了插入、删除、替换、分割和填充字符串等运算，以及编码和格式化字符串的方法。

```
protected void Page_Load(object sender, EventArgs e)
{
    ///测试插入运算
    InsertString();Response.Write("<hr /><br />");
    ///测试删除运算
    RemoveString();Response.Write("<hr /><br />");
    ///测试替换运算
    ReplaceString();Response.Write("<hr /><br />");
    ///测试分割运算
    SplitString();Response.Write("<hr /><br />");
    ///测试填充运算
    PadString();Response.Write("<hr /><br />");
    ///测试比较运算
    CompareString();Response.Write("<hr /><br />");
    ///测试连接运算
    JoinString();Response.Write("<hr /><br />");
    ///处理字符串的空白
    BlankString();Response.Write("<hr /><br />");
    ///大小写转换
    UpperLitterString();Response.Write("<hr /><br />");
    ///匹配字符串
    MatchString();Response.Write("<hr /><br />");
    ///获取子字符串
    SubString();Response.Write("<hr /><br />");
    ///格式化字符串
    FormatString();Response.Write("<hr /><br />");
    ///字符串编码
    EncodingString();
}
```

把 StringDealWith.aspx 页面设置为起始页面，并运行 StringApplication 程序，结果如图 7.3 所示。

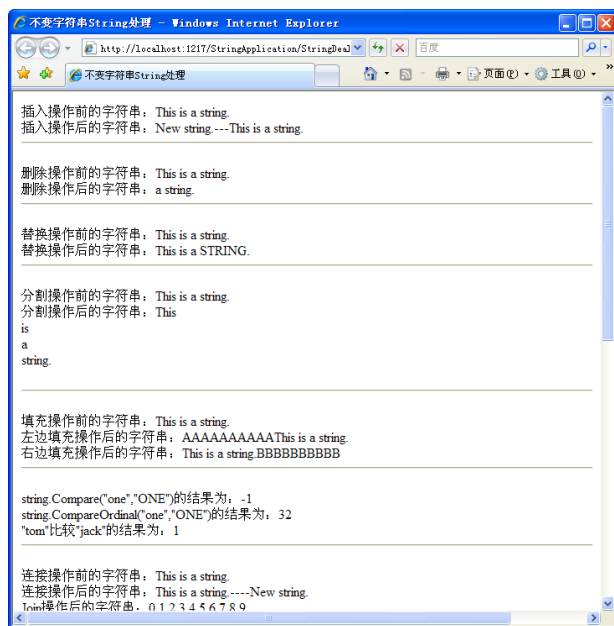


图 7.3 StringDealWith.aspx 页面测试不可变字符串 String 处理的方法

7.2 8 种可变字符串 `StringBuilder` 处理

`StringBuilder` 类表示可变字符串，它能够创建动态修改的字符串对象。其中，动态修改字符串对象的运算包括追加、移除、替换和插入等。本节将介绍动态字符串的处理方法。

7.2.1 `StringBuilder` 类和对象

`StringBuilder` 类表示可变字符串。使用 `StringBuilder` 类不但可以创建可变的字符串对象或实例，而且还可以使用追加、移除、替换和插入等运算来修改这些实例。另外，这些实例还可以使用 `MaxCapacity` 属性指定实例的最大容量。默认容量为 16，默认的最大容量是 `Int32.MaxValue`。`StringBuilder` 类可以按存储字符的需要分配更多的内存，同时对容量进行相应的调整。`StringBuilder` 类提供了 4 个常用属性，具体说明如表 7.2 所示。

表 7.2 `StringBuilder` 类的属性

属性	说明
<code>Length</code>	实例的长度
<code>Capacity</code>	当前实例所分配的最大字符数，它的值一般大于 <code>Length</code>
<code>MaxCapacity</code>	实例的最大容量
<code>Chars</code>	获取指定字符位置处的字符

在创建 `StringBuilder` 类的实例时，可以指定 `Capacity` 和 `MaxCapacity` 属性的值。`Capacity` 属性的值根据实例的长度动态改变。如果实例添加字符串之后的长度大于 `MaxCapacity` 属性的值，则程序会抛出异常。

7.2.2 追加字符串

向 `StringBuilder` 类的实例追加字符串可以使用 `Append()`、`AppendLine()` 和 `AppendFormat()` 方法。上述三种方法都是把指定的对象（如字符、字符串等）追加到 `StringBuilder` 类实例的末尾。如果被追加的对象不是字符串形式，则上述方法将该对象自动转换为字符串形式。其中，`AppendLine()` 方法可以追加一个换行符号，`AppendFormat()` 方法可以设置追加字符串的格式。

`Append()` 方法的重载方法比较多，共有 19 种，具体说明如下。

- ❑ `Append(bool value)`: 追加指定布尔值的字符串形式，`value` 参数表示追加的布尔值。
- ❑ `Append(byte value)`: 追加指定 8 位无符号整数的字符串形式，`value` 参数表示追加的 8 位无符号整数。
- ❑ `Append(char value)`: 追加指定 Unicode 字符的字符串形式，`value` 参数表示追加的 Unicode 字符。
- ❑ `Append(char[] value)`: 追加指定字符数组的字符串形式，`value` 参数表示追加的字符数组。
- ❑ `Append(decimal value)`: 追加指定十进制数的字符串形式，`value` 参数表示追加的十进制数。
- ❑ `Append(double value)`: 追加指定双精度浮点数的字符串形式，`value` 参数表示追加的双精度浮点数。
- ❑ `Append(short value)`: 追加指定 16 位有符号整数的字符串形式，`value` 参数表示追加的 16 位有符号整数。
- ❑ `Append(int value)`: 追加指定 32 位有符号整数的字符串形式，`value` 参数表示追加的 32 位有符号整数。

- ❑ `Append(long value)`: 追加指定 64 位有符号整数的字符串形式, `value` 参数表示追加的 64 位有符号整数。
- ❑ `Append(Object value)`: 追加指定对象的字符串形式, `value` 参数表示追加的对象。
- ❑ `Append(sbyte value)`: 追加指定 8 位有符号整数的字符串形式, `value` 参数表示追加的 8 位有符号整数。
- ❑ `Append(float value)`: 追加指定单精度浮点数的字符串形式, `value` 参数表示追加的单精度浮点数。
- ❑ `Append(string value)`: 追加指定字符串, `value` 参数表示追加的字符串。
- ❑ `Append(ushort value)`: 追加指定 16 位无符号整数的字符串形式, `value` 参数表示追加的 16 位无符号整数。
- ❑ `Append(uint value)`: 追加指定 32 位无符号整数的字符串形式, `value` 参数表示追加的 32 位无符号整数。
- ❑ `Append(ulong value)`: 追加指定 64 位无符号整数的字符串形式, `value` 参数表示追加的 64 位无符号整数。
- ❑ `Append(char value,int repeatCount)`: 追加指定字符的字符串形式, `value` 参数表示追加的字符, `repeatCount` 参数表示追加 `value` 参数的次数。
- ❑ `Append(char[] value,int startIndex,int charCount)`: 追加指定字符数组的字符串形式, `value` 参数表示追加的字符数组, `startIndex` 参数表示 `value` 参数中的起始位置, `charCount` 参数表示要追加的字符数量。
- ❑ `Append(string value,int startIndex,int count)`: 追加指定的字符串, `value` 参数表示追加的字符串, `startIndex` 参数表示 `value` 参数中的起始位置, `count` 参数表示要追加的字符数量。

`AppendFormat()`方法向 `StringBuilder` 类的实例追加内容时, 可以指定格式化字符串的内容。该方法有 5 种重载方式, 具体说明如下。

- ❑ `AppendFormat(string format,Object arg0)`: 追加一个格式化对象。
- ❑ `AppendFormat(string format,params Object[] args)`: 追加格式化的对象数组。
- ❑ `AppendFormat(IFormatProvider provider,string format,params Object[] args)`: 追加格式化的动态对象数组。
- ❑ `AppendFormat(string format,Object arg0,Object arg1)`: 追加两个格式化对象。
- ❑ `AppendFormat(string format,Object arg0,Object arg1,Object arg2)`: 追加三个格式化对象。

其中, `format` 参数指定格式化字符串, `arg0`、`arg1` 和 `arg2` 参数都表示追加的对象, `args` 表示追加的对象数组, `provider` 参数指定 `format` 参数中格式设置规范的解释方式。

`AppendLine()`方法可以向 `StringBuilder` 类的实例追加换行符号, 并且该换行符号追加在实例的末尾。该方法有两种重载方式, 具体说明如下。

- ❑ `AppendLine()`: 追加一个换行符号。
- ❑ `AppendLine(string value)`: 追加一个字符串 (由 `value` 参数指定), 并在末尾追加一个换行符号。

下面的代码使用了 `Append()`、`AppendLine()`和 `AppendFormat()`方法向 `StringBuilder` 类的实例 `sb` 追加了变量 `str1`、变量 `str2`、“----|----”、当前时间的格式化字符串、换行符号、“----结束----”和“New Line...”等字符串。最后, `AppendString(string str1,string str2)`函数返回实例 `sb` 的字符串

形式。

```

/// <summary>
/// 追加字符串
/// </summary>
/// <param name="str1">参数 1</param>
/// <param name="str2">参数 2</param>
/// <returns></returns>
private string AppendString(string str1,string str2)
{
    StringBuilder sb = new StringBuilder();
    ///追加字符串
    sb.Append(str1);
    sb.Append("----|----");
    sb.Append(str2);
    ///追加当前日期，并格式化
    sb.AppendFormat("{0:d}",DateTime.Now);
    ///追加换行符
    sb.AppendLine();
    ///追加结束行
    sb.AppendLine("----结束----");
    ///追加新行
    sb.Append("New Line...");
    ///返回一个String 对象
    return sb.ToString().Replace("\n","<br />");
}

```

7.2.3 插入字符串

Insert()方法能够将指定对象的字符串表示形式插入到 `StringBuilder` 类的实例中指定字符位置。Insert()方法的重载方法比较多，共有 18 种，具体说明如下。

- ❑ `public StringBuilder Insert(int index,bool value)`: 插入布尔值的字符串形式，`value` 参数表示插入的布尔值。
- ❑ `public StringBuilder Insert(int index,byte value)`: 插入 8 位无符号整数的字符串形式，`value` 参数表示插入的 8 位无符号整数。
- ❑ `public StringBuilder Insert(int index,char value)`: 插入字符的字符串形式，`value` 参数表示插入的字符。
- ❑ `public StringBuilder Insert(int index,char[] value)`: 插入字符数组的字符串形式，`value` 参数表示插入的字符数组。
- ❑ `public StringBuilder Insert(int index,decimal value)`: 插入十进制数的字符串形式，`value` 参数表示插入的十进制数。
- ❑ `public StringBuilder Insert(int index,double value)`: 插入双精度浮点数的字符串形式，`value` 参数表示插入的双精度浮点数。
- ❑ `public StringBuilder Insert(int index,short value)`: 插入 16 位有符号整数的字符串形式，`value` 参数表示插入的 16 位有符号整数。
- ❑ `public StringBuilder Insert(int index,int value)`: 插入 32 位有符号整数的字符串形式，`value` 参数表示插入的 32 位有符号整数。
- ❑ `public StringBuilder Insert(int index,long value)`: 插入 64 位有符号整数的字符串形式，`value` 参数表示插入的 64 位有符号整数。

- ❑ `public StringBuilder Insert(int index, Object value)`: 插入对象的字符串形式, `value` 参数表示插入的对象。
- ❑ `public StringBuilder Insert(int index, sbyte value)`: 插入 8 位有符号整数的字符串形式, `value` 参数表示插入的 8 位有符号整数。
- ❑ `public StringBuilder Insert(int index, float value)`: 插入单精度浮点数的字符串形式, `value` 参数表示插入的单精度浮点数。
- ❑ `public StringBuilder Insert(int index, string value)`: 插入字符串, `value` 参数表示插入的字符串。
- ❑ `public StringBuilder Insert(int index, ushort value)`: 插入 16 位无符号整数的字符串形式, `value` 参数表示插入的 16 位无符号整数。
- ❑ `public StringBuilder Insert(int index, uint value)`: 插入 32 位无符号整数的字符串形式, `value` 参数表示插入的 32 位无符号整数。
- ❑ `public StringBuilder Insert(int index, ulong value)`: 插入 64 位无符号整数的字符串形式, `value` 参数表示插入的 64 位无符号整数。
- ❑ `public StringBuilder Insert(int index, string value, int count)`: 插入字符串, `value` 参数表示插入的布尔值, `count` 参数表示插入的次数。
- ❑ `public StringBuilder Insert(int index, char[] value, int startIndex, int charCount)`: 插入字符数组的字符串形式, `value` 参数表示插入的字符数组, `startIndex` 参数表示字符数组的开始位置, `charCount` 参数表示插入的字符数量。

上述方法中, `index` 参数表示插入的位置。

下面的代码使用了 `Insert()` 方法向 `StringBuilder` 类的实例 `sb` 插入了变量 `str1`、变量 `str2`、“----|----”、当前时间的字符串和“----结束----”等字符串。最后, `InsertString(string str1, string str2)` 函数返回实例 `sb` 的字符串形式。

```
/// <summary>
/// 插入字符串
/// </summary>
/// <param name="str1">参数 1</param>
/// <param name="str2">参数 2</param>
/// <returns></returns>
private string InsertString(string str1, string str2)
{
    StringBuilder sb = new StringBuilder();
    ///插入字符串
    sb.Insert(0, str1);
    sb.Insert(sb.Length, "----|----");
    sb.Insert(sb.Length, str2);
    ///插入时间字符串
    sb.Insert(0, DateTime.Now.ToString() + " ");
    sb.Insert(sb.Length, "----结束----");
    ///返回一个 String 对象
    return sb.ToString().Replace("\n", "<br />");
}
```

7.2.4 替换字符串

`Replace()` 方法能够将 `StringBuilder` 类的实例中指定字符或字符串替换为其他的指定字符或字

字符串。该方法有 4 种重载形式，具体说明如下。

- ❑ `public StringBuilder Replace(char oldChar,char newChar)`: 将指定的字符替换为其他字符。
- ❑ `public StringBuilder Replace(string oldValue,string newValue)`: 将指定的字符串替换为其他字符串。
- ❑ `public StringBuilder Replace(char oldChar,char newChar,int startIndex,int count)`: 将指定的字符替换为其他字符，并指定开始位置和替换长度。
- ❑ `public StringBuilder Replace(string oldValue,string newValue,int startIndex,int count)`: 将指定的字符串替换为其他字符串，并指定开始位置和替换长度。

其中，`oldChar` 参数为被替换的字符，`newChar` 参数为替换后的字符，`oldValue` 参数为被替换的字符串，`newValue` 参数为替换后的字符串，`startIndex` 参数指定替换的开始位置，`count` 参数指定替换的长度。

下面的代码首先创建了一个 `StringBuilder` 类的实例 `sb`，并向 `sb` 中追加了由数字组成的字符串。然后，使用 `Replace()` 方法首先把字符串“00”替换为字符串“aa”、再把字符“0”替换为字符“A”。`ReplaceString(int max)` 函数还使用 `Response.Write()` 方法分别在网页上显示了替换运算之前和之后的字符串。最后，该函数返回实例 `sb` 的字符串形式。

```

/// <summary>
/// 替换字符串
/// </summary>
/// <param name="max">参数 1</param>
/// <returns></returns>
private string ReplaceString(int max)
{
    ///构造一个可变字符串对象
    StringBuilder sb = new StringBuilder();
    for(int i = 0; i < max; i++)
    {
        sb.Append(i.ToString().PadLeft(3,'0'));
    }
    ///输出字符串
    Response.Write("未替换运算之前的字符串: " + sb.ToString() + "<br />");
    ///替换字符串
    sb.Replace("00","aa");
    ///输出字符串
    Response.Write("替换运算之后字符串: " + sb.ToString() + "<br />");
    ///替换字符串
    sb.Replace('0','A');
    ///输出字符串
    Response.Write("替换运算之后字符串: " + sb.ToString() + "<br />");
    ///返回一个 String 对象
    return sb.ToString().Replace("\n","<br />");
}

```

7.2.5 删除字符串

`Remove()` 方法能够从 `StringBuilder` 类的实例中移除指定范围的字符或字符串。它的原形如下。

```
public StringBuilder Remove(int startIndex,int length)
```

其中，`startIndex` 参数指定开始移除的位置；`length` 参数指定要移除的字符数量。

下面的代码首先创建了一个 `StringBuilder` 类的实例 `sb`，并向 `sb` 中追加了由数字组成的字符

串。然后，使用了 `Remove()` 方法移除从位置 0 开始的、实例 `sb` 的一半内容。`RemoveString(int max)` 函数还使用 `Response.Write()` 方法分别在网页上显示了移除运算之前和之后的字符串。最后，该函数返回实例 `sb` 的字符串形式。

```
/// <summary>
/// 删除字符串
/// </summary>
/// <param name="max">参数 1</param>
/// <returns></returns>
private string RemoveString(int max)
{
    ///构造一个可变字符串对象
    StringBuilder sb = new StringBuilder();
    for(int i = 0; i < max; i++)
    {
        sb.Append(i.ToString().PadLeft(3, '0'));
    }
    ///输出字符串
    Response.Write("未删除运算之前的字符串: " + sb.ToString() + "<br />");
    ///移除一半的字符串
    sb.Remove(0, max * 3 / 2);
    ///输出字符串
    Response.Write("删除运算之后的字符串: " + sb.ToString() + "<br />");
    ///返回一个 String 对象
    return sb.ToString().Replace("\n", "<br />");
}
```

7.2.6 复制字符串

复制 `StringBuilder` 类实例的内容可以使用 `CopyTo()` 方法。该方法可以将此实例指定段中的字符复制到目标字符数组的指定段中。它的原形如下。

```
public void CopyTo(int sourceIndex, char[] destination, int destinationIndex, int count)
```

其中，参数 `sourceIndex` 指定开始复制字符的位置，`destination` 参数指定要将字符复制到的字符数组，`destinationIndex` 参数指定要将字符复制到的 `destination` 参数中的起始位置，`count` 参数指定要复制的字符数。

下面的代码首先创建了一个 `StringBuilder` 类的实例 `sb`，并向 `sb` 中追加了由数字组成的字符串。然后，创建了一个保存复制结果的字符数组 `destChar`，并使用 `CopyTo()` 方法把实例 `sb` 的内容全部复制到字符数组 `destChar`。`CopyString(int max)` 函数还使用了 `Response.Write()` 方法分别在网页上显示了复制运算之前和之后的字符串。最后，该函数返回实例 `sb` 的字符串形式。

```
/// <summary>
/// 复制字符串
/// </summary>
/// <param name="max">参数 1</param>
/// <returns></returns>
private string CopyString(int max)
{
    ///构造一个可变字符串对象
    StringBuilder sb = new StringBuilder();
    for(int i = 0; i < max; i++)
    {
        sb.Append(i.ToString().PadLeft(3, '0'));
    }
    ///输出字符串
```

```

Response.Write("未复制运算之前的字符串: " + sb.ToString() + "<br />");
///复制字符串
char[] destChar = new char[sb.Length];
sb.CopyTo(0,destChar,0,sb.Length);
///输出字符串
Response.Write("输出复制运算之后字符串: ");
for(int i = 0; i < destChar.Length; i++)
{
Response.Write(destChar[i].ToString());
}
Response.Write("<br />");
///返回一个 String 对象
return sb.ToString().Replace("\n", "<br />");
}

```

7.2.7 处理字符串容量

EnsureCapacity()方法能够确保 StringBuilder 类实例的容量至少是指定值（即 capacity 参数的值）。该方法的原形如下。

```
public int EnsureCapacity(int capacity)
```

其中，capacity 参数指定要确保的最小容量。

下面的代码首先创建了一个 StringBuilder 类的实例 sb，并使用 capacity 和 maxCapacity 参数指定了该实例的初始化大写和最大容量。然后，使用 EnsureCapacity()方法确保实例 sb 的最小容量为 10。随后，使用 for 语句向实例 sb 中追加字符串，并在追加运算之前判断是否超过实例 sb 的最大容量。最后，StringCapacity(int capacity,int maxCapacity)函数返回实例 sb 的字符串形式。

```

/// <summary>
/// 处理字符串容量
/// </summary>
/// <param name="capacity"></param>
/// <param name="maxCapacity">最大容量</param>
/// <returns></returns>
private string StringCapacity(int capacity,int maxCapacity)
{
    ///创建限制容量的可变字符串对象
    StringBuilder sb = new StringBuilder(capacity,maxCapacity);
    ///确保最小容量为 10
    sb.EnsureCapacity(10);
    for(int i = 0; i < maxCapacity + 1; i++)
    {
        ///保证不超过最大容量
        if(sb.Length + i.ToString().Length < maxCapacity){sb.Append(i);}
    }
    return sb.ToString();
}

```

7.2.8 可变字符串 StringBuilder 处理的应用

ASP.NET 应用程序 StringBuilderApplication 演示了可变字符串 StringBuilder 的处理方法。该应用程序添加了名称为 DealWithStringBuilder.aspx 的页面，并测试了本节中的追加、插入、删除、替换和复制字符串，以及处理字符串容量的方法。另外，StringBuilder 类包含在名字空间 System.Text 中，因此需要引入该名字空间，程序代码如下。

```

///引入命名空间
using System.Text;

```

下面的代码是 DealWithStringBuilder.aspx.cs 文件的部分内容。该文件在 Page_Load(object

sender, EventArgs e)事件中测试了追加、插入、删除、替换和复制字符串，以及处理字符串容量的方法。

```
public partial class DealWithStringBuilder: System.Web.UI.Page
{
protected void Page_Load(object sender, EventArgs e)
{
    ///测试追加字符串
    Response.Write(AppendString("aaaaa", "bbbbbb") + "<br />");
    ///测试插入字符串
    Response.Write(InsertString("aaaaa", "bbbbbb") + "<br />");
    ///测试移除字符串
    RemoveString(10);
    ///测试替换字符串
    ReplaceString(100);
    ///测试复制字符串
    CopyString(10);
    ///处理字符串容量
    int maxCapacity = 100;
    Response.Write("最大容量不超过" + maxCapacity.ToString() + ": ");
    Response.Write(StringCapacity(0, maxCapacity));
}
}
///以下代码省略
}
```

把 DealWithStringBuilder.aspx 页面设置为起始页面，并运行 StringBuilderApplication 程序，结果如图 7.4 所示。

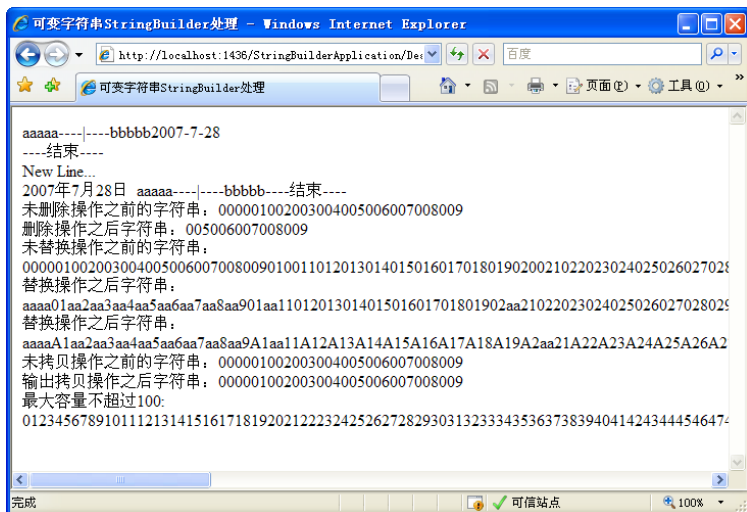


图 7.4 DealWithStringBuilder.aspx 页面测试可变字符串 StringBuilder 处理的方法

第 8 章 常见的.NET 框架中正则表达式及其应用

.NET 框架（Framework）提供了专门用来表示正则表达式的类——Regex。该类可以使用正则表达式来处理字符串，如检测给定的字符串是否匹配给定的正则表达式、替换正则表达式匹配的字符串、使用正则表达式来分割给定的字符串等。另外，使用该类还可以验证用户输入的数据或信息是否满足给定的要求，从而在一定程度上提高应用程序的安全性。

本章内容将介绍使用正则表达式类 Regex 来验证、匹配、替换、分组和分割给定字符串的方法，以及使用正则表达式验证给定的字符串是否满足给定要求。最后，还介绍了使用正则表达式从网页中提取网页标题、图像地址和 HTTP 地址等信息的方法。

8.1 10 种.NET 框架中的正则表达式类库

本节介绍.NET 框架中处理正则表达式的类、枚举和委托等。这些类、枚举或委托都被包含在名字空间 System.Text.RegularExpressions 中。因此，要使用这些类、枚举或委托时，往往需要引用该名字空间。名字空间 System.Text.RegularExpressions 中的 Regex 类表示正则表达式，它能够使用正则表达式来处理字符串，如匹配、替换和分割字符串等。

8.1.1 System.Text.RegularExpressions 命名空间

System.Text.RegularExpressions 命名空间提供了正则表达式功能，它包含了一些类、枚举或委托等对象。它们的具体说明如表 8.1 所示。

表 8.1 System.Text.RegularExpressions命名空间中的类、枚举和委托

类、枚举和委托	说明
Regex	不可变的正则表达式
Match	单个正则表达式匹配的结果
MatchCollection	匹配结果的集合
Capture	单个成功捕获中的一个子字符串
CaptureCollection	由单个捕获组执行的捕获的集合
Group	单个捕获组的结果
GroupCollection	单个匹配中的捕获组的集合
RegexOptions（枚举）	正则表达式选项的枚举值
MatchEvaluator（委托）	在Replace方法运算过程中，处理每个匹配的委托
RegexCompilationInfo	正则表达式的编译信息
RegexRunner	编译正则表达式的基类
RegexRunnerFactory	为编译过的正则表达式提供创建RegexRunner类的功能

8.1.2 正则表达式类 Regex

Regex 类位于 System.Text.RegularExpressions 命名空间之中，表示不可变（只读）正则表达式类。该类可以使用正则表达式处理给定的字符串。特别地，该类还提供了多个静态方法。因此，在不创建类的实例情况下，可以使用这些静态方法来处理给定的字符串。其中，Regex 类提供的属性如表 8.2 所示、Regex 类的静态方法如表 8.3 所示、Regex 类的实例方法如表 8.4 所示。

表 8.2 Regex类的属性

属性	说明
CacheSize（静态）	当前Regex对象的缓存中已编译正则表达式的最大项数
Options	Regex实例的正则表达式选项
RightToLeft	表示正则表达式是否从右向左进行搜索

CacheSize 属性是一个静态属性。一般情况下，应用程序不需要设置或修改该属性的值，而是由 Regex 类内部维护。

表 8.3 Regex类的静态方法

静态方法	说明
CompileToAssembly()	编译正则表达式，并保存到磁盘中
Escape()	通过替换为转义字符来转义最小的元字符集
Unescape()	取消输入字符串中的转义字符

CompileToAssembly()静态方法能够编译给定的正则表达式，并存放指定的程序集中。Escape()静态方法能够转换字符串，并使得这些字符串能够在正则表达式中作为常数使用。

表 8.4 Regex类的实例方法

实例方法	说明
IsMatch()	表示正则表达式是否在输入字符串中找到匹配项，即是否匹配
Match()	匹配给定的正则表达式，并返回一个匹配项
Matches()	匹配给定的正则表达式，并返回多个匹配项
Replace()	用指定的字符串替换所有匹配项
Split()	在匹配的位置将输入字符串拆分为一个子字符串数组
GetGroupNames()	获取正则表达式的捕获组名数组
GetGroupNumbers()	获取与数组中的组名相对应的捕获组号的数组
GroupNameFromNumber()	获取与指定组号相对应的组名
GroupNumberFromName()	返回与指定组名相对应的组号

注意：虽然表 8.4 中的方法均为实例方法，但是 Regex 类能够在不创建类的实例情况下，就使用 IsMatch()、Match()、Matches()、Replace()和 Split()方法来处理给定的字符串。

8.1.3 正则表达式选项

开发人员在使用 Regex 类或创建该类的实例时，都可以使用正则表达式选项。该选项是一个类型为 RegexOptions 的枚举值。RegexOptions 枚举提供了用于设置正则表达式选项的枚举值，具体说明如表 8.5 所示。

表 8.5 正则表达式选项

选项	说明
None	不设置选项
IgnoreCase	不区分大小写的匹配
IgnorePatternWhitespace	消除正则表达式中的非转义空白，并启用由#标记的注释
Singleline	单行模式
Multiline	多行模式，更改^和\$的含义
RightToLeft	搜索从右向左而不是从左向右进行
Compiled	将正则表达式编译为程序集
ExplicitCapture	有效地捕获形式为(?<name>...)的显式命名或编号的组
ECMAScript	启用符合ECMAScript的行为
CultureInvariant	忽略语言中的区域性差异

8.1.4 检查是否匹配表达式

Regex 类的 IsMatch()方法表示可以检查正则表达式在输入字符串中是否找到匹配项，即输入字符串是否匹配给定的正则表达式。如果找到匹配项，则返回 true，否则返回 false。IsMatch()方法有以下 4 种重载方法。

- ❑ Regex.IsMatch(string input)
- ❑ Regex.IsMatch(string input, int startat)
- ❑ Regex.IsMatch(string input,string pattern)
- ❑ Regex.IsMatch(string input,string pattern,RegexOptions options)

其中，input 参数指定输入字符串，pattern 参数指定正则表达式，startat 参数指定开始搜索的字符位置，options 参数指定匹配选项。

下面的代码检查在字符串“0123456789”中是否找到正则表达式“\d+”的匹配项。其中，Regex.IsMatch()方法使用 Regex 类的 IsMatch()静态方法；IsMatch()方法创建一个 Regex 实例 regex，并使用该实例的 IsMatch()实例方法。

```

/// <summary>
/// 检查是否匹配
/// </summary>
/// <returns></returns>
private bool RegexIsMatch()
{
    string input = "0123456789";
    string pattern = @"\d+";
    return Regex.IsMatch(input,pattern);
}
/// <summary>
/// 检查是否匹配
/// </summary>
/// <returns></returns>
private bool IsMatch()
{
    string input = "0123456789";
    string pattern = @"\d+";
    Regex regex = new Regex(pattern);
    return regex.IsMatch(input);
}

```

下面的代码是检查在字符串“ABCDEFGF”中是否找到正则表达式“[a-z]+”的匹配项。另外，

在检查过程中启用了 `RegexOptions.IgnoreCase` 选项。其中，`Regex.IsMatchWithOptions()` 方法使用 `Regex` 类的 `IsMatch()` 静态方法；`IsMatchWithOptions()` 方法创建一个 `Regex` 实例 `regex`，并使用该实例的 `IsMatch()` 实例方法。

```
/// <summary>
/// 检查是否匹配，并带有选项
/// </summary>
/// <returns></returns>
private bool RegexIsMatchWithOptions()
{
    string input = "ABCDEFGH";
    string pattern = @"[a-z]+";
    return Regex.IsMatch(input, pattern, RegexOptions.IgnoreCase);
}
/// <summary>
/// 检查是否匹配，并带有选项
/// </summary>
/// <returns></returns>
private bool IsMatchWithOptions()
{
    string input = "ABCDEFGH";
    string pattern = @"[a-z]+";
    Regex regex = new Regex(pattern, RegexOptions.IgnoreCase);
    return regex.IsMatch(input);
}
```

8.1.5 匹配单个匹配项

`Regex` 类的 `Match()` 方法可以在输入字符串中、根据给定的正则表达式找到匹配项，并把找到的匹配项作为单个匹配项返回。其中，每一个匹配项都为 `Match` 类型。

`Match` 类表示单个正则表达式匹配的结果，它包含两个属性和两个方法。其中，`Empty` 属性表示空组、`Groups` 属性表示匹配的组的集合、`NextMatch()` 方法获取从当前匹配结束位置开始的下一个匹配结果、`Result()` 方法返回已传递的替换模式的扩展。

`Regex` 类的 `Match()` 方法有以下 5 种重载方法。

- ❑ `Regex.Match(string input)`
- ❑ `Regex.Match(string input, int startat)`
- ❑ `Regex.Match(string input, string pattern)`
- ❑ `Regex.Match(string input, int beginning, int length)`
- ❑ `Regex.Match(string input, string pattern, RegexOptions options)`

其中，`input` 参数指定输入字符串，`pattern` 参数指定正则表达式，`startat` 参数指定开始搜索的字符位置，`options` 参数指定匹配选项，`beginning` 参数指定在输入字符串中开始搜索的字符位置，`length` 参数指定子字符串中包含在搜索中的字符数。

下面的代码是在字符串“0123456789”中查找正则表达式“\d+”的匹配项。其中，`RegexMatch()` 方法使用 `Regex` 类的 `Match()` 静态方法；`Match()` 方法创建一个 `Regex` 实例 `regex`，并使用该实例的 `Match()` 实例方法。

```
/// <summary>
/// 匹配给定的表达式
/// </summary>
/// <returns></returns>
private string RegexMatch()
```

```

{
    string input = "0123456789";
    string pattern = @"\d+";
    Match match = Regex.Match(input, pattern);
    if (match != null) { return match.Value; }
    return string.Empty;
}
/// <summary>
/// 匹配给定的表达式
/// </summary>
/// <returns></returns>
private string Match()
{
    string input = "0123456789";
    string pattern = @"\d+";
    Regex regex = new Regex(pattern);
    Match match = regex.Match(input);
    if (match != null) { return match.Value; }
    return string.Empty;
}

```

下面的代码是在字符串“ABCDEFGH”中查找正则表达式“[a-z]+”的匹配项。另外，在查找过程中启用了 `RegexOptions.IgnoreCase` 选项。其中，`RegexMatchOptions()` 方法使用 `Regex` 类的 `Match()` 静态方法；`MatchOptions()` 方法创建一个 `Regex` 实例 `regex`，并使用该实例的 `Match()` 实例方法。

```

/// <summary>
/// 匹配给定的表达式，并带有选项
/// </summary>
/// <returns></returns>
private string RegexMatchOptions()
{
    string input = "ABCDEFGH";
    string pattern = @"[a-z]+";
    Match match = Regex.Match(input, pattern, RegexOptions.IgnoreCase);
    if (match != null) { return match.Value; }
    return string.Empty;
}
/// <summary>
/// 匹配给定的表达式，并带有选项
/// </summary>
/// <returns></returns>
private string MatchOptions()
{
    string input = "ABCDEFGH";
    string pattern = @"[a-z]+";
    Regex regex = new Regex(pattern, RegexOptions.IgnoreCase);
    Match match = regex.Match(input);
    if (match != null) { return match.Value; }
    return string.Empty;
}

```

8.1.6 匹配多个匹配项

`Regex` 类的 `Matches()` 方法可以在输入字符串中、根据给定的正则表达式找到匹配项，并把找到的匹配项作为单个匹配项返回。其中，每一个匹配项也都为 `Match` 类型。`Regex` 类的 `Matches()` 方法有以下 4 种重载方法。

- ❑ `Regex.Matches(string input)`
- ❑ `Regex.Matches(string input, int startat)`
- ❑ `Regex.Matches(string input, string pattern)`
- ❑ `Regex.Matches(string input, string pattern, RegexOptions options)`

其中，`input` 参数指定输入字符串，`pattern` 参数指定正则表达式，`startat` 参数指定开始搜索的字符位置，`options` 参数指定匹配选项。

下面的代码是在字符串“0123456789abcd321bfr987”中查找正则表达式“\d+”的匹配项。其中，`RegexMatches()`方法使用 `Regex` 类的 `Matches()`静态方法；`Matches()`方法创建一个 `Regex` 实例 `regex`，并使用该实例的 `Matches()`实例方法。

```

/// <summary>
/// 匹配给定的表达式
/// </summary>
/// <returns></returns>
private string[] RegexMatches()
{
    string input = "0123456789abcd321bfr987";
    string pattern = @"\d+";
    MatchCollection matches = Regex.Matches(input, pattern);
    if(matches == null) return null;
    string[] result = new string[matches.Count];
    for(int i = 0; i < result.Length; i++)
    {
        result[i] = matches[i].Value;
    }
    return result;
}
/// <summary>
/// 匹配给定的表达式
/// </summary>
/// <returns></returns>
private string[] Matches()
{
    string input = "0123456789abcd321bfr987";
    string pattern = @"\d+";
    Regex regex = new Regex(pattern);
    MatchCollection matches = regex.Matches(input);
    if(matches == null) return null;
    string[] result = new string[matches.Count];
    for(int i = 0; i < result.Length; i++)
    {
        result[i] = matches[i].Value;
    }
    return result;
}

```

下面的代码是在字符串“abcdABCDeDfgEDFGwyz”中查找正则表达式“[a-z]+”的匹配项。另外，在查找过程中启用了 `RegexOptions.IgnoreCase` 选项。其中，`RegexMatches()`方法使用 `Regex` 类的 `Matches()`静态方法；`Matches()`方法创建一个 `Regex` 实例 `regex`，并使用该实例的 `Matches()`实例方法。

```

/// <summary>
/// 匹配给定的表达式，并带有选项
/// </summary>
/// <returns></returns>
private string[] RegexMatchesOptions()

```

```

{
    string input = "abcdABCDedfgEDFGwyz";
    string pattern = @"[a-z]+";
    MatchCollection matches
    = Regex.Matches(input,pattern,RegexOptions.IgnoreCase);
    if(matches == null) return null;
    string[] result = new string[matches.Count];
    for(int i = 0; i < result.Length; i++)
    {
        result[i] = matches[i].Value;
    }
    return result;
}
/// <summary>
/// 匹配给定的表达式，并带有选项
/// </summary>
/// <returns></returns>
private string[] MatchesOptions()
{
    string input = "abcdABCDedfgEDFGwyz";
    string pattern = @"[a-z]+";
    Regex regex = new Regex(pattern,RegexOptions.IgnoreCase);
    MatchCollection matches = regex.Matches(input);
    if(matches == null) return null;
    string[] result = new string[matches.Count];
    for(int i = 0; i < result.Length; i++)
    {
        result[i] = matches[i].Value;
    }
    return result;
}

```

8.1.7 替换

Regex 类的 Replace()方法可以在输入的字符串中、根据给定的正则表达式找到匹配项，并把找到的匹配项作为单个匹配项返回。Regex 类的 Replace()方法有以下 10 种重载方法。

- ❑ Regex.Replace(string input,MatchEvaluator evaluator)
- ❑ Regex.Replace(string input,string replacement)
- ❑ Regex.Replace(string input,MatchEvaluator evaluator,int count)
- ❑ Regex.Replace(string input,string replacement,int count)
- ❑ Regex.Replace(string input,string pattern,MatchEvaluator evaluator)
- ❑ Regex.Replace(string input,string pattern,string replacement)
- ❑ Regex.Replace(string input,MatchEvaluator evaluator,int count,int startat)
- ❑ Regex.Replace(string input,string replacement,int count,int startat)
- ❑ Regex.Replace(string input,string pattern,MatchEvaluator evaluator,RegexOptions options)
- ❑ Regex.Replace(string input,string pattern,string replacement,RegexOptions options)

其中，input 参数指定输入字符串，evaluator 参数指定处理匹配项的委托，replacement 参数指定替换的字符串，count 参数指定替换的最大次数，pattern 参数指定正则表达式，startat 参数指定开始搜索的字符位置，options 参数指定匹配选项。

下面的代码是在字符串“0123456789abcd321bfr987”中查找正则表达式“\d+”的匹配项，并将匹配项替换为字符串“xyz”。其中，RegexReplace()方法使用 Regex 类的 Replace()静态方法；

Replace()方法是创建一个 Regex 实例 regex，并使用该实例的 Replace()实例方法。

```
/// <summary>
/// 替换匹配的表达式
/// </summary>
/// <returns></returns>
private string RegexReplace()
{
    string input = "0123456789abcd321bfr987";
    string pattern = @"\d+";
    string replacement = "xyz";
    return Regex.Replace(input, pattern, replacement);
}
/// <summary>
/// 替换匹配的表达式
/// </summary>
/// <returns></returns>
private string Replace()
{
    string input = "0123456789abcd321bfr987";
    string pattern = @"\d+";
    string replacement = "xyz";
    Regex regex = new Regex(pattern);
    return regex.Replace(input, replacement);
}
```

8.1.8 使用委托 MatchEvaluator 处理匹配结果

当 Regex 类的 Replace()方法替换匹配项时，还可以使用 MatchEvaluator 委托处理每一个匹配项。下面的代码定义了一个 MatchEvaluator 委托的方法，并处理匹配项 m，即返回由该匹配项的值及其长度组成的字符串。

```
/// <summary>
/// 处理匹配的选项
/// </summary>
/// <param name="m"></param>
/// <returns></returns>
private string ReplaceMatchEvaluator(Match m)
{
    return m.Value + m.Length.ToString();
}
```

下面的代码是在字符串“01abcd231bfr987”中查找正则表达式“\d+”的匹配项，并使用委托 evaluator 处理每一个匹配项（处理匹配项的方法为 ReplaceMatchEvaluator(Match m)）。其中，RegexReplacewithEvaluator()方法使用 Regex 类的 Replace()静态方法；ReplacewithEvaluator()方法创建一个 Regex 实例 regex，并使用该实例的 Replace()实例方法。

```
/// <summary>
/// 替换匹配的表达式，并使用委托 MatchEvaluator 处理匹配的表达式
/// </summary>
/// <returns></returns>
private string RegexReplacewithEvaluator()
{
    string input = "01abcd231bfr987";
    string pattern = @"\d+";
    MatchEvaluator evaluator = new MatchEvaluator(ReplaceMatchEvaluator);
    return Regex.Replace(input, pattern, evaluator);
}
/// <summary>
```

```

/// 替换匹配的表达式, 并使用委托 MatchEvaluator 处理匹配的表达式
/// </summary>
/// <returns></returns>
private string ReplaceWithEvaluator()
{
    string input = "01abcd231bfr987";
    string pattern = @"\"d+";
    Regex regex = new Regex(pattern);
    MatchEvaluator evaluator = new MatchEvaluator(ReplaceMatchEvaluator);
    return regex.Replace(input, evaluator);
}

```

8.1.9 获取分组名称

Regex 类的 GroupName()方法可以获取其实例的正则表达式中组的名称。下面的代码获取了正则表达式 “\w*(?<char>\w+)\k<char>” 组的名称, 并返回一个字符串数组。

```

/// <summary>
/// 获取分组的名称
/// </summary>
/// <returns></returns>
private string[] GroupName()
{
    string pattern = @"\"w*(?<char>\w+)\k<char>";
    Regex regex = new Regex(pattern);
    return regex.GetGroupNames();
}

```

8.1.10 分割表达式

Regex 类的 Split()方法可以根据给定的正则表达式找到匹配项, 并把输入字符串分割为字符串数组。Regex 类的 Split()方法有以下 5 种重载方法。

- ❑ Regex.Split(string input);
- ❑ Regex.Split(string input,int count);
- ❑ Regex.Split(string input,string pattern);
- ❑ Regex.Split(string input,int count,int startat);
- ❑ Regex.Split(string input,string pattern,RegexOptions options);

其中, input 参数指定输入字符串, count 参数指定要返回的最大数组元素数, pattern 参数指定正则表达式, startat 参数指定开始搜索的字符位置, options 参数指定匹配选项。

下面的代码是使用正则表达式 “\d+” 的匹配项分割字符串 “A0123456789abcd321bfr987Z”, 并返回分割后的字符串数组。其中, RegexSplit()方法使用 Regex 类的 Split()静态方法; Split()方法创建一个 Regex 实例 regex, 并使用该实例的 Split()实例方法。

```

/// <summary>
/// 分割匹配的表达式
/// </summary>
/// <returns></returns>
private string[] RegexSplit()
{
    string input = "A0123456789abcd321bfr987Z";
    string pattern = @"\"d+";
    return Regex.Split(input, pattern);
}
/// </summary>

```

```

/// 分割匹配的表达式
/// </summary>
/// <returns></returns>
private string[] Split()
{
    string input = "A0123456789abcd321bfr987Z";
    string pattern = @"\d+";
    Regex regex = new Regex(pattern);
    return regex.Split(input);
}

```

8.2 14 种正则表达式类 Regex 处理字符串

本节介绍使用正则表达式类 **Regex** 处理字符串，以及提取网页中内容的方法。其中，使用正则表达式类 **Regex** 可以验证整数、实数、电话号码、邮政编码、身份证号码、银行卡号、日期和时间、IP 地址、车牌号码、电子邮件和 URL 等。提取网页中的内容包括提取网页标题、提取网页中的 HTTP 地址和提取网页中的图像地址。

8.2.1 只包含数字验证

只包含数字的字符串可以使用元字符 **\d** 来验证。以下正则表达式能够验证最小长度为零的只包含数字的字符串。

```
\d*
```

使用上述正则表达式后，下面的代码能够验证只包含数字的字符串。其中，**pattern** 变量定义了正则表达式，**input** 变量定义了被验证的只包含数字的字符串。如果验证成功，则 **Regex.IsMatch(input,pattern)** 函数返回 **true**，否则返回 **false**。

```

string pattern = @"\d*";
string input = "0123456789";
Regex.IsMatch(input,pattern);

```

8.2.2 整数验证

整数包含正整数、0 和负整数。因此，在验证整数时需要验证这 3 种情况。以下正则表达式能够验证整数（包含正整数、0 和负整数）。

```
(0|-?[1-9]\d*)
```

使用上述正则表达式后，下面的代码能够验证整数（包含正整数、0 和负整数）。其中，**pattern** 变量定义了正则表达式，**input** 变量定义了被验证的整数（包含正整数、0 和负整数）字符串。如果验证成功，则 **Regex.IsMatch(input,pattern)** 函数返回 **true**，否则返回 **false**。

```

string pattern = @"(0|-?[1-9]\d*)";
string input = "123456789";
Regex.IsMatch(input,pattern);

```

8.2.3 实数验证

实数包含正整数、0、负整数、正小数和负小数。因此，在验证实数时需要验证这 5 种情况。以下正则表达式能够验证实数（包含正整数、0、负整数、正小数和负小数）。

```
-?(0|([1-9]\d*))(\.\d+)?
```

使用上述正则表达式后，下面的代码能够验证实数（包含正整数、0、负整数、正小数和负

小数)。其中, `pattern` 变量定义了正则表达式, `input` 变量定义了被验证的实数(包含正整数、0、负整数、正小数和负小数)字符串。如果验证成功, 则 `Regex.IsMatch(input,pattern)` 函数返回 `true`, 否则返回 `false`。

```
string pattern = @"-?(0|([1-9]\d*))(\.\d+)?";
string input = "123456789.001002003";
Regex.IsMatch(input,pattern);
```

8.2.4 电话号码验证

电话号码一般由区号、固定电话号码、分机号码, 以及它们之间的连接符号组成。其中, 区号、分机号码和连接符号是可选部分。以下正则表达式能够验证电话号码。

```
(0\d{2,3}(\<char>[- ])?)\d{7,8}(\k<char>\d{4})?
```

使用上述正则表达式后, 下面的代码能够验证电话号码(包括区号、固定电话号码、分机号码和连接符号)。其中, `pattern` 变量定义了正则表达式, `input` 变量定义了被验证的电话号码。如果验证成功, 则 `Regex.IsMatch(input,pattern)` 函数返回 `true`, 否则返回 `false`。

```
string pattern = @"(0\d{2,3}(\<char>[- ])?)\d{7,8}(\k<char>\d{4})?";
string input = "010-66668888-1234";
Regex.IsMatch(input,pattern);
```

8.2.5 邮政编码验证

邮政编码一般为长度为 6 的数字字符串。以下正则表达式能够验证邮政编码。

```
\d{6}
```

使用上述正则表达式后, 下面的代码能够验证邮政编码(其中, 字符串长度为 6)。其中, `pattern` 变量定义了正则表达式, `input` 变量定义了被验证的邮政编码。如果验证成功, 则 `Regex.IsMatch(input,pattern)` 函数返回 `true`, 否则返回 `false`。

```
string pattern = @"\d{6}";
string input = "100001";
Regex.IsMatch(input,pattern);
```

8.2.6 身份证号码验证

身份证号码一般为长度为 15 或 18 位的数字字符串。其中, 前面的 17 位由数字组成, 最后一位可以为数字也可以为字母 X 或 x。以下正则表达式能够验证身份证号码。

```
(\d{17}(\d|x|X))|\d{15}
```

使用上述正则表达式后, 下面的代码能够验证身份证号码(包括 15 位和 18 位)。其中, `pattern` 变量定义了正则表达式, `input` 变量定义了被验证的身份证号码。如果验证成功, 则 `Regex.IsMatch(input,pattern)` 函数返回 `true`, 否则返回 `false`。

```
string pattern = @"(\d{17}(\d|x|X))|\d{15}";
string input = "100102198008080118";
Regex.IsMatch(input,pattern);
```

8.2.7 银行卡号验证

例如, 某银行卡号由 13~19 位数字组成, 组成结构为: “9XXXXX X……X X”。前面 6 位为发卡行标识代码, 第一位固定为 9, 后面 5 位由 BIN 注册管理机构分配。接着是 6~12 位的自定义位, 它发卡行自己定义。最后是 1 位校验数字验码。以下正则表达式能够验证银行卡号。

```
9\d{12,18}
```

使用上述正则表达式后，下面的代码能够验证银行卡号（13~19 位）。其中，`pattern` 变量定义了正则表达式，`input` 变量定义了被验证的银行卡号。如果验证成功，则 `Regex.IsMatch(input,pattern)` 函数返回 `true`，否则返回 `false`。

```
string pattern = @"9\d{12,18}";
string input = "9558000011112222888";
Regex.IsMatch(input,pattern);
```

8.2.8 IP 地址验证

IP 地址组成规则为：1~3 位整数.1~3 位整数.1~3 位整数.1~3 位整数，且每一个整数都要小于或等于 255。以下正则表达式能够验证 IP 地址。

```
((25[0-5])|(2[0-4]\d)|(1\d{2})|([1-9]\d?|0))\.){3}((25[0-5])|(2[0-4]\d)|(1\d{2})|([1-9]\d?|0))
```

使用上述正则表达式后，下面的代码能够验证 IP 地址。其中，`pattern` 变量定义了正则表达式，`input` 变量定义了被验证的 IP 地址。如果验证成功，则 `Regex.IsMatch(input,pattern)` 函数返回 `true`，否则返回 `false`。

```
string pattern = @"(((25[0-5])|(2[0-4]\d)|(1\d{2})|([1-9]\d?|0))\.){3}((25[0-5])|(2[0-4]\d)|(1\d{2})|([1-9]\d?|0))";
string input = "188.188.210.221";
Regex.IsMatch(input,pattern);
```

8.2.9 日期和时间验证

日期和时间字符串为长格式的日期（包含年月日）和时间（24 小时制）字符串。它的格式为“yyyy-MM-dd hh:mm:ss”。年、月、日的字符串的长度为 4、2、2，它们之间的连接符号可以是 -、/ 或（空格）等。小时、分和秒的字符串的长度都为 2，且使用冒号（:）分割。另外，小时应在 0~23 范围之内，分和秒都应在 0~59 范围之内。以下正则表达式能够验证日期和时间字符串。

```
\d{4}(?<cc>[-./ ])(1[0-2])|(0?\d))\k<cc>((12)\d)|(3[01])|(0?\d))
```

使用上述正则表达式后，下面的代码能够验证日期和时间字符串（包括年、月、日、时、分、秒）。其中，`pattern` 变量定义了正则表达式，`input` 变量定义了被验证的日期和时间字符串。如果验证成功，则 `Regex.IsMatch(input,pattern)` 函数返回 `true`，否则返回 `false`。

```
string pattern = @"^\d{4}(?<cc>[-./ ])(1[0-2])|(0?\d))\k<cc>((12)\d)|(3[01])|(0?\d))";
string input = "100102198008080118";
Regex.IsMatch(input,pattern);
```

8.2.10 车牌号码验证

通用车牌号码的组成为：直辖市或省的简称+1 个大写英文字母+1 个大写英文字母或 1 个数字+1 个大写英文字母或 1 个数字+3 个数字。以下正则表达式能够验证车牌号码。

```
\w[A-Z][A-Z0-9]{2}\d{3}
```

使用上述正则表达式后，下面的代码能够验证车牌号码（包括直辖市或省的简称）。其中，`pattern` 变量定义了正则表达式，`input` 变量定义了被验证的车牌号码。如果验证成功，则 `Regex.IsMatch(input,pattern)` 函数返回 `true`，否则返回 `false`。

```
string pattern = @"^\w[A-Z][A-Z0-9]{2}\d{3}";
string input = "京A88888";
Regex.IsMatch(input,pattern);
```

8.2.11 电子邮件验证

电子邮件是当前网络时代最常用的应用之一。邮件地址一般由名称、字符@、域名后缀组成,如 admin@admin.com、123_d@123.com 等。以下正则表达式能够验证电子邮件。

```
\w+([-+.']\w+)*@\w+([-.\w+)*\.\w+([-.\w+)*
```

使用上述正则表达式后,下面的代码能够验证电子邮件。其中, pattern 变量定义了正则表达式, input 变量定义了被验证的电子邮件。如果验证成功,则 Regex.IsMatch(input,pattern)函数返回 true, 否则返回 false。

```
string pattern = @"^[\w+([-+.']\w+)*@\w+([-.\w+)*\.\w+([-.\w+)*";
string input = "admin@admin.com";
Regex.IsMatch(input,pattern);
```

8.2.12 URL 验证

URL, 即 HTTP 地址, 一般是以字符串“http://”或“https://”开头的字符串。它可以被字符、/、?、&、%和=分割。如 http://www.ab.com、http://cn.net/2007/07/06/aa.aspx?ID=1 等。以下正则表达式能够验证 HTTP 地址。

```
http(s)?://([a-z-]+\.)+[\w-]+(/[\w- ./?%&=]*)?
```

使用上述正则表达式后,下面的代码能够验证 HTTP 地址(包含了不是以 HTTP 开头的地址)。其中, pattern 变量定义了正则表达式, input 变量定义了被验证的 HTTP 地址。如果验证成功,则 Regex.IsMatch(input,pattern)函数返回 true, 否则返回 false。

```
string pattern = @"http(s)?://([a-z-]+\.)+[\w-]+(/[\w- ./?%&=]*)?";
string input = "http://www.ab.com";
Regex.IsMatch(input,pattern);
```

8.2.13 提取网页标题

为了显示提取网页的标题,特创建了 ASP.NET 应用程序 RegexApplication, 并且还为其添加了 GetPageHtmlData.aspx 页面。该页面实现了提取网页标题、HTTP 地址和图像地址等功能。

GetPageHtmlData.aspx 页面添加了 1 个 TextBox 控件、1 个非空验证控件、1 个正则表达式验证控件和 3 个 Button 控件。它们的 ID 属性值分别为 tbUrl、rfUrl、reUrl、btnGetTitle、btnGetUrl 和 btnGetImage。GetPageHtmlData.aspx 页面的设计界面如图 8.1 所示。

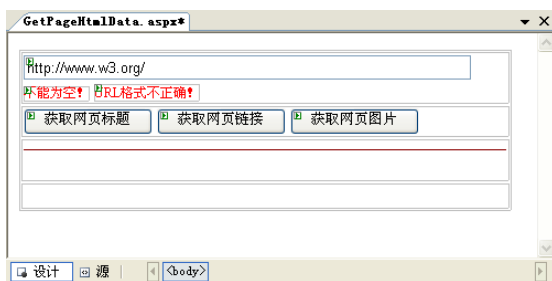


图 8.1 GetPageHtmlData.aspx 页面的设计界面

GetPageHtmlData.aspx 页面的部分 HTML 代码如下:

```
<asp:TextBox ID="tbUrl" runat="server"
Width="400px">http://www.w3.org/</asp:TextBox>
<asp:RequiredFieldValidator ID="rfUrl" runat="server"
ControlToValidate="tbUrl" Display="Dynamic" ErrorMessage="不能为空!"
Font-Size="9pt"></asp:RequiredFieldValidator>
```

```
<asp:RegularExpressionValidator ID="reUrl" runat="server"
ControlToValidate="tbUrl" Display="Dynamic"
ErrorMessage="URL 格式不正确！" Font-Size="9pt"
ValidationExpression="http(s)?://([\\w-]+\\.)+[\\w-]+(/[\\w- ./?%&=]*)?">
</asp:RegularExpressionValidator></td>
<asp:Button ID="btnGetTitle" runat="server" Text="获取网页标题"
OnClick="btnGetTitle_Click" />
<asp:Button ID="btnGetUrl" runat="server" Text="获取网页链接"
OnClick="btnGetUrl_Click" />
<asp:Button ID="btnGetImage" runat="server" Text="获取网页图片"
OnClick="btnGetImage_Click" />
```

在下面的代码示例中，`GetPageData(string url)`函数根据给定的 URL 获取该页面的内容。它首先创建 `WebClient` 类的示例 `client`，并设置该示例的编码，最后调用 `DownloadString()`方法获取网页的内容。

```
private string GetPageData(string url)
{
    ///创建 WebClient 对象
    WebClient client = new WebClient();
    ///设置编码
    client.Encoding = System.Text.Encoding.Default;
    ///获取网页的内容
    return(client.DownloadString(url));
}
```

`btnGetTitle` 控件定义了 `Click` 事件 `btnGetTitle_Click(object sender,EventArgs e)`。该事件获取网站 (`www.w3c.org`) 的标题。实现的具体步骤如下。

- ① 定义获取网页标题的正则表达式 `(?<=<title\s*[^\>]*>).*?(?=<\title>)`。
- ② 调用函数 `GetPageData(string url)`，获取网页 `www.w3c.org` 的内容，并保存在变量 `input` 中。
- ③ 调用 `Regex.Match()`方法，从变量 `input` 中匹配到网页标题，并把匹配结果保存在变量 `match` 中。
- ④ 显示匹配的网页标题。

```
protected void btnGetTitle_Click(object sender,EventArgs e)
{
    ///定义获取标题的正则表达式
    string pattern = @"(?<=<title\s*[^\>]*>).*?(?=<\title>)";
    ///获取网页内容
    string input = GetPageData(tbUrl.Text.Trim());
    ///获取匹配的标题
    Match match = Regex.Match(input,pattern,RegexOptions.IgnoreCase);
    if(match != null)
    {
        ///显示标题
        Data = "网页标题为：" + match.Value;
    }
}
```

把 `GetPageHtmlData.aspx` 页面设置为起始页面，并运行应用程序 `RegexApplication`。单击“获取网页标题”按钮，如图 8.2 所示。

8.2.14 提取网页中的图像地址

`btnGetImage` 控件定义了 `Click` 事件 `btnGetImage_Click(object sender,EventArgs e)`。该事件获取网站 (`www.w3c.org`) 的所有

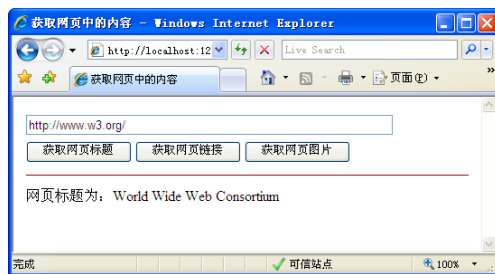


图 8.2 单击“获取网页标题”按钮

图像地址。具体实现步骤如下。

- ❶ 定义获取网页标题的正则表达式 `(?<=src\s*=\s*["']?)http(s)?://([\w-]+\.)+[\w-]+(/[\w- ./?%&=]*)?`。
- ❷ 调用函数 `GetPageData(string url)`，获取网页 `www.w3c.org` 的内容，并保存在变量 `input` 中。
- ❸ 调用 `Regex.Matches()` 方法，从变量 `input` 中匹配所有的图像地址，并将匹配结果保存在变量 `matches` 中。
- ❹ 依次处理变量 `matches` 中的每一个匹配项，并为每一个匹配项创建一个 HTML 元素 ``，以使用图片形式显示该匹配项。
- ❺ 显示匹配的图像地址。

```
protected void btnGetImage_Click(object sender, EventArgs e)
{
    ///定义获取图片的链接地址的正则表达式
    string pattern = @"(?<=src\s*=\s*[" + @"'" + 
        + @"'" + "])\s*http(s)?://([\w-]+\.)+[\w-]+(/[\w- ./?%&=]*)?";
    ///获取网页内容
    string input = GetPageData(tbUrl.Text.Trim());
    ///获取匹配的图像的链接地址
    MatchCollection matches
    = Regex.Matches(input, pattern, RegexOptions.IgnoreCase);
    if(matches != null)
    {
        ///创建 HTML 链接元素<img>
        StringBuilder sb = new StringBuilder();
        foreach(Match match in matches)
        {
            sb.Append("<img src=\"" + match.Value + "\" />");
            sb.Append("<br />");
        }
        ///显示链接地址
        Data = "图片如下: <br />" + sb.ToString();
    }
}
```

把 `GetPageHtmlData.aspx` 页面设置为起始页面，并运行应用程序 `RegexApplication`。单击“获取网页图片”按钮，如图 8.3 所示。

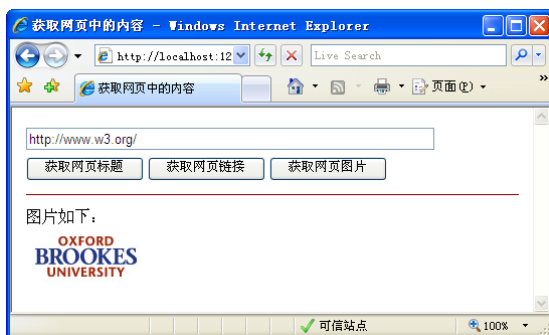


图 8.3 提取网页中的图像地址

8.2.15 提取网页中的 HTTP 地址

`btnGetUrl` 控件定义了 `Click` 事件 `btnGetUrl_Click(object sender, EventArgs e)`。该事件获取网站 (`www.w3c.org`) 的所有 HTTP 地址。具体实现步骤如下。

- ❶ 定义获取网页标题的正则表达式 `http(s)?://([\w-]+\.)+[\w-]+(/[\w- ./?%&=]*)?`。

- ② 调用函数 `GetPageData(string url)`，获取网页 `www.w3c.org` 的内容，并保存在变量 `input` 中。
- ③ 调用 `Regex.Matches()` 方法，从变量 `input` 中匹配所有的 HTTP 地址，并将匹配结果保存在变量 `matches` 中。
- ④ 依次处理变量 `matches` 中的每一个匹配项，并为每一个匹配项创建一个 HTML 元素 `<a>`，以使用链接形式显示该匹配项。
- ⑤ 显示匹配的 HTTP 地址。

```
protected void btnGetUrl_Click(object sender, EventArgs e)
{
    ///定义获取链接地址的正则表达式
    string pattern = @"http(s)?://([\w-]+\.)+[\w-]+(/[\w- ./?%&=]*)?";
    ///获取网页内容
    string input = GetPageData(tbUrl.Text.Trim());
    ///获取匹配的链接地址
    MatchCollection matches
    = Regex.Matches(input, pattern, RegexOptions.IgnoreCase);
    if(matches != null)
    {
        ///创建 HTML 链接元素<a>
        StringBuilder sb = new StringBuilder();
        foreach(Match match in matches)
        {
            sb.Append("<a href=\"" + match.Value + "\">"
            + match.Value + "</a>");
            sb.Append("<br />");
        }
        ///显示链接地址
        Data = "网页链接地址如下: <br />" + sb.ToString();
    }
}
```

把 `GetPageHtmlData.aspx` 页面设置为起始页面，并运行应用程序 `RegexApplication`。单击“获取网页链接”按钮，如图 8.4 所示。



图 8.4 提取网页中的 HTTP 地址

第 9 章 常见 ASP.NET 验证控件

ASP.NET 提供了一组验证控件，并提供了一种易用但功能强大的检错方式，同时在必要时还可以向用户显示错误信息。ASP.NET 验证不但能够检查用户输入的信息是否有效，还可以减少网站受到攻击的可能性。ASP.NET 验证提供了非空验证、范围验证、比较验证、正则表达式验证和用户自定义验证等多种验证方式，以及用于处理或集中显示页面上所有验证控件的提示或错误信息功能。

注意：虽然 ASP.NET 验证控件是服务器端控件，但是验证运算一般都在客户端发生（除非定义了特定的服务器端运算，如 CustomValidator 控件的服务器端验证），即不需要提交到服务器。

9.1 ASP.NET 验证简介

安全是一个永恒的话题，几乎所有的网站或 Web 应用程序都提供了各种各样的方式接受用户的输入。然而，并不是每一个用户输入的信息都是安全的。特别地，一些恶意用户故意输入一些信息，导致网站或 Web 应用程序的稳定性和安全性受到损害。为了提高应用程序的安全性，ASP.NET 也提供了验证用户输入的功能，以及与之相关的服务器端控件。其中，ASP.NET 提供了以下 5 个服务器端验证控件。

- ❑ RequiredFieldValidator 控件，验证用户输入的值不能等于事先设定的初始值。如果该初始值为空，则是非空验证，即验证用户输入不能为空。
- ❑ RangeValidator 控件，验证用户输入信息在一个指定的范围之内。
- ❑ CompareValidator 控件，比较两个控件的内容是否满足指定的关系，或者比较某个控件的内容和指定的值是否满足指定的关系。
- ❑ RegularExpressionValidator 控件，验证用户输入的值是否符合指定的正则表达式。
- ❑ CustomValidator 控件，开发人员可以通过客户端函数或服务器端函数来定义或指定验证规则。

上述 5 个控件都派生于 BaseValidator 类。该类定义了 ASP.NET 验证控件所共有的属性，如用来显示错误消息的 ErrorMessage、指定被验证控件 ID 的 ControlToValidate 等属性。BaseValidator 类的属性及其说明如表 9.1 所示。

表 9.1 BaseValidator类的属性及其说明

属性	说明
ControlToValidate	被验证控件的ID
ErrorMessage	验证失败后，显示的错误信息
Text	验证失败后，显示的文本信息，它的优先级高于ErrorMessage
Display	显示错误信息的方式，它的值可以为None、Static或Dynamic
ValidationGroup	被验证控件所属的验证组的名称

续表

属性	说明
ForeColor	错误信息的颜色
Enabled	表示控件是否可用
EnableClientScript	表示是否启用客户端验证

另外，ASP.NET 还提供了一个与验证相关的控件——ValidationSummary。它可以用来处理或集中显示页面上所有验证控件的错误信息。ASP.NET 中与验证相关的控件之间的继承关系如图 9.1 所示。

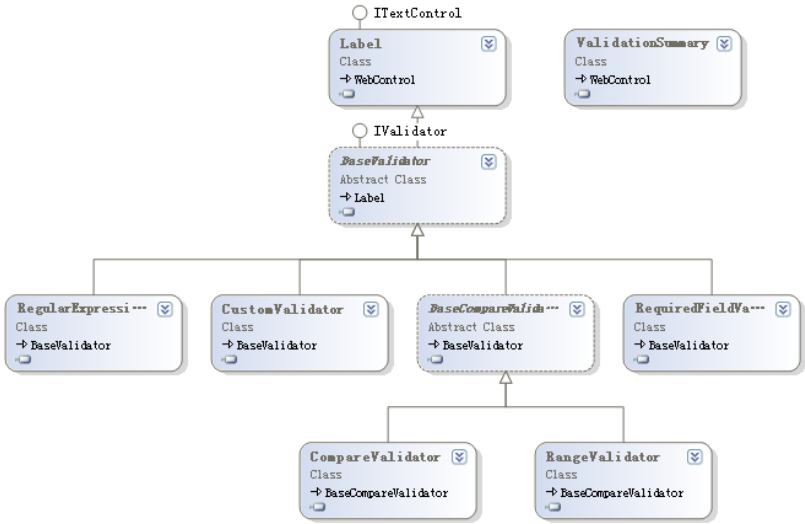


图 9.1 ASP.NET 中与验证相关的控件之间的继承关系

对于每一种 ASP.NET 验证控件而言，最常用的属性为：ErrorMessage 和 Display。其中，ErrorMessage 属性用来保存控件验证失败之后所显示的提示或错误信息。Display 属性用来指定控件在网页上显示的方式，它的值可以为 None、Static 和 Dynamic。这 3 个值分别表示验证控件不同的显示方式，具体说明如下。

- ❑ None，不显示方式，即验证控件不显示提示或错误信息。此时，验证控件不占有网页上的空间。特别地，如果要把所有单个验证控件的提示或错误信息集中使用 ValidationSummary 控件来显示，则需要把单个的验证控件的 Display 属性设置为该值。
- ❑ Static，静态显示方式（系统默认值）。此时，验证控件将始终占用网页上的一部分空间。如果用户输入的值通过验证，则验证控件将不显示提示或错误信息，但是验证控件仍然占用网页上的一部分空间。
- ❑ Dynamic，动态显示方式。只有当验证控件显示提示或错误消息时，它才占用网页上的一部分空间。否则，它将不占用网页上的空间。特别地，当一个输入控件同时被几个验证控件验证时，往往使用该方式来设置每一个验证控件，这样可以提高页面的美观效果。

9.2 2 种非空验证

RequiredFieldValidator 控件能够验证用户输入的值不能等于事先设定的初始值（由 InitialValue

属性指定)。如果该初始值为空,则是非空验证,即验证用户输入不能为空。如果该初始值不为空(即是一个指定的值),那么验证用户输入的值将不能等于该指定的值。

注意: RequiredFieldValidator 控件的 InitialValue 属性的默认值为空字符串,即该控件默认为非空验证控件。

9.2.1 无初始值的非空验证

无初始值的非空验证由 RequiredFieldValidator 控件实现。此时,该控件 InitialValue 属性的值为空字符串。因此,被验证控件的内容不能等于空字符串,从而实现非空验证的功能。验证效果如图 9.2 所示。

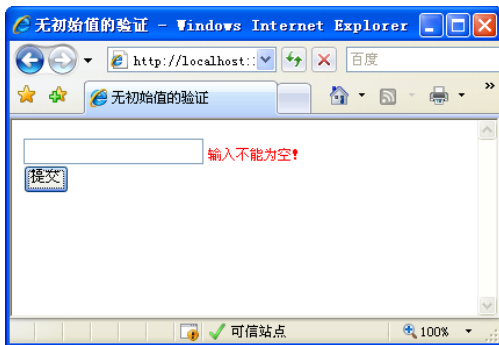


图 9.2 无初始值的非空验证

下面的代码使用了 RequiredFieldValidator 验证控件 rfValue 来验证 tbValue 控件的内容是否为空。如果该内容为空,则 rfValue 控件显示提示信息“输入不能为空!”。

注意: 用户在 tbValue 控件中输入了内容,并在焦点离开该控件后,rfValue 控件将验证 tbValue 控件的内容,并同时显示验证结果(如果验证失败,则显示验证控件的提示信息)。

```
<!-- ASPNETValidator/NotNull.aspx -->
<%@ Page Language="C#" AutoEventWireup="true" %>
<html xmlns="http://www.w3.org/1999/xhtml" >
<head runat="server"><title>无初始值的验证</title></head>
<body><form id="form1" runat="server">
<asp:TextBox ID="tbValue" runat="server"></asp:TextBox>
<asp:RequiredFieldValidator ID="rfValue" runat="server"
ErrorMessage="输入不能为空!" Font-Size="9pt"
ControlToValidate="tbValue"></asp:RequiredFieldValidator>
<br />
<asp:Button ID="btnCheck" runat="server" Text="提交" />
</form></body>
</html>
```

9.2.2 指定初始值的验证

如果一旦指定了 RequiredFieldValidator 控件的 InitialValue 属性的值,那么该控件将验证被验证的内容(往往是输入框中的内容)和 InitialValue 属性的值是否相等。如果不相等,则验证成功,否则验证失败,同时显示提示信息。验证效果如图 9.3 所示。

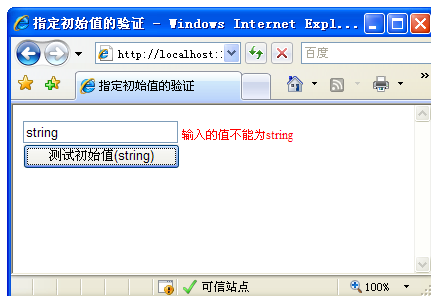


图 9.3 指定初始值的验证

下面的代码使用了 `RequiredFieldValidator` 验证控件 `rfValue` 来验证 `tbValue` 控件的内容不能等于它的 `InitialValue` 属性的值“string”。如果 `tbValue` 控件的内容等于“string”，则 `rfValue` 控件显示提示信息“输入的值不能为 string”。

```
<!-- ASPNETValidator/NotNullInitValue.aspx -->
<%@ Page Language="C#" AutoEventWireup="true" %>
<html xmlns="http://www.w3.org/1999/xhtml" >
<head runat="server"><title>指定初始值的验证</title></head>
<body><form id="form1" runat="server">
<asp:TextBox ID="tbValue" runat="server"></asp:TextBox>
<asp:RequiredFieldValidator ID="rfValue" runat="server"
ErrorMessage="输入的值不能为 string" InitialValue="string"
Font-Size="9pt"
ControlToValidate="tbValue"></asp:RequiredFieldValidator>
<br />
<asp:Button ID="btnCheck" runat="server" Text="提交" />
</form></body>
</html>
```

9.3 3 种范围验证

`RangeValidator` 控件能够验证用户输入的值是否在一个指定的范围之内。如果用户输入的值不在该范围之内，则显示提示信息。`RangeValidator` 控件提供了 5 种范围，由 `Type` 属性指定。该属性的值的具体描述如下。

- ❑ `String`: 字符串范围。
- ❑ `Integer`: 整数范围。
- ❑ `Double`: 实数范围。
- ❑ `Date`: 日期范围。
- ❑ `Currency`: 与货币相关的范围。

上述 5 种范围的最大值由 `MaximumValue` 属性指定、最小值由 `MinimumValue` 属性指定，且最大值必须要大于或等于最小值。

9.3.1 整数范围验证

当 `RangeValidator` 控件的 `Type` 属性的值为“`Integer`”时，它将指定一个整数范围。此时，它可以验证用户输入的值是否在这个整数范围之内。如果用户输入的值在该范围之内，则验证成功，否则验证失败，同时显示提示信息。验证效果如图 9.4 所示。



图 9.4 整数范围验证

下面的代码使用了 RangeValidator 控件 rvValue 来验证 tbValue 控件的内容是否在指定的整数范围 (0~100) 之内。如果该内容不在指定范围之内, 则显示提示信息“输入的数值必须在 0~100 之间。”。另外, 该示例还使用了 RequiredFieldValidator 验证控件 rfValue 来验证 tbValue 控件的内容是否为空。如果该内容为空, 则 rfValue 控件显示提示信息“输入不能为空!”。

```
<!-- ASPNETValidator/RangeInt.aspx -->
<%@ Page Language="C#" AutoEventWireup="true" %>
<html xmlns="http://www.w3.org/1999/xhtml" >
<head runat="server"><title>整数范围验证</title></head>
<body><form id="form1" runat="server">
<asp:TextBox ID="tbValue" runat="server"></asp:TextBox>
<asp:RequiredFieldValidator ID="rfValue" runat="server"
ErrorMessage="输入不能为空! " Font-Size="9pt" ControlToValidate="tbValue"
Display="Dynamic"></asp:RequiredFieldValidator>
<asp:RangeValidator ID="rvValue" runat="server"
ControlToValidate="tbValue" ErrorMessage="输入的数值必须在 0~100 之间。"
Font-Size="9pt" MaximumValue="100" MinimumValue="0" Type="Integer"
Display="Dynamic"></asp:RangeValidator>
<br />
<asp:Button ID="btnCheck" runat="server" Text="提交" />
</form></body>
</html>
```

9.3.2 字母范围验证

当 RangeValidator 控件的 Type 属性的值为“String”时, 它将指定一个字符串范围。此时, 它可以验证用户输入的值是否在这个字符串范围之内。如果用户输入的值在该范围之内, 则验证成功, 否则验证失败, 同时显示提示信息。验证效果如图 9.5 所示。

注意: 字母范围属于字符串范围。因此, 图 9.5 中的 RangeValidator 控件的 Type 属性的值为“String”。

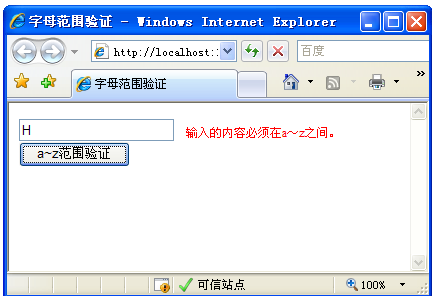


图 9.5 字母范围验证

下面的代码使用了 RangeValidator 控件 rvValue 来验证 tbValue 控件的内容是否在指定的字母范围（a—z）之内。如果该内容不在指定范围之内，则显示提示信息“输入的内容必须在 a—z 之间。”。另外，该示例还使用了 RequiredFieldValidator 验证控件 rfValue 来验证 tbValue 控件的内容是否为空。如果该内容为空，则 rfValue 控件显示提示信息“输入不能为空！”。

```
<!-- ASPNETValidator/RangeLetter.aspx -->
<%@ Page Language="C#" AutoEventWireup="true" %>
<html xmlns="http://www.w3.org/1999/xhtml" >
<head runat="server"><title>字母范围验证</title></head>
<body><form id="form1" runat="server">
<asp:TextBox ID="tbValue" runat="server"></asp:TextBox>
<asp:RequiredFieldValidator ID="rfValue" runat="server"
ErrorMessage="输入不能为空！" Font-Size="9pt" ControlToValidate="tbValue"
Display="Dynamic"></asp:RequiredFieldValidator>
<asp:RangeValidator ID="rvValue" runat="server"
ControlToValidate="tbValue" ErrorMessage="输入的内容必须在 a—z 之间。"
Font-Size="9pt" MaximumValue="z" MinimumValue="a"
Display="Dynamic"></asp:RangeValidator>
<br />
<asp:Button ID="btnCheck" runat="server" Text="提交" />
</form></body>
</html>
```

9.3.3 日期范围验证

当 RangeValidator 控件的 Type 属性的值为“Date”时，它将指定一个日期范围。此时，它可以验证用户输入的值是否在这个日期范围之内。如果用户输入的值在该范围之内，则验证成功，否则验证失败，同时显示提示信息。验证效果如图 9.6 所示。

注意：图 9.6 中的 RangeValidator 控件的 Type 属性的值为“Date”。



图 9.6 日期范围验证

下面的代码使用了 RangeValidator 控件 rvValue 来验证 tbValue 控件的内容是否在指定的日期范围（2007-01-01~2007-08-01）之内。如果该内容不在指定范围之内，则显示提示信息“输入的内容必须在 2007-01-01~2007-08-01 之间。”。另外，该示例还使用了 RequiredFieldValidator 验证控件 rfValue 来验证 tbValue 控件的内容是否为空。如果该内容为空，则 rfValue 控件显示提示信息“输入不能为空！”。

```
<!-- ASPNETValidator/RangeDate.aspx -->
<%@ Page Language="C#" AutoEventWireup="true" %>
<html xmlns="http://www.w3.org/1999/xhtml" >
<head runat="server"><title>日期范围验证</title></head>
<body><form id="form1" runat="server">
```

```

<asp:TextBox ID="tbValue" runat="server"></asp:TextBox>
<asp:RequiredFieldValidator ID="rfValue" runat="server"
ErrorMessage="输入不能为空！" Font-Size="9pt" ControlToValidate="tbValue"
Display="Dynamic"></asp:RequiredFieldValidator>
<asp:RangeValidator ID="rvValue" runat="server"
ControlToValidate="tbValue"
ErrorMessage="输入的日期必须在 2007-01-01~2007-08-01 之间。"
Font-Size="9pt" MaximumValue="2007-08-01" MinimumValue="2007-01-01"
Type="Date" Display="Dynamic"></asp:RangeValidator>
<br />
<asp:Button ID="btnCheck" runat="server"
Text="2007-01-01~2007-08-01 之间验证" />
</form></body>
</html>

```

9.4 3 种比较验证

CompareValidator 控件能够进行以下 3 种类型的验证。

- ❑ 比较两个控件的内容满足指定的关系。
- ❑ 比较一个控件和指定的值满足指定的关系。
- ❑ 验证控件的内容是否为指定的数据类型。

CompareValidator 控件的 ControlToValidate 属性指定被验证的控件、ControlToCompare 属性指定被比较的控件、ValueToCompare 属性指定被比较的值。其中，Operator 属性指定被验证的控件或值所满足的关系，它的值的具体描述如下。

- ❑ Equal: 相等关系，系统默认值。
- ❑ NotEqual: 不相等关系。
- ❑ GreaterThan: 大于关系。
- ❑ GreaterThanEqual: 大于或等于关系。
- ❑ LessThan: 小于关系。
- ❑ LessThanEqual: 小于或等于关系。
- ❑ DataTypeCheck: 验证控件的内容是否为指定的数据类型。该数据类型由 Type 属性指定。

注意：上述关系中的左运算数（关系的左边内容）为验证控件的内容，右运算数为比较控件的内容或指定的值。

另外，CompareValidator 控件和 RangeValidator 控件一样，也提供了 5 种数据类型，并由 Type 属性指定。该属性的值的具体描述如下。

- ❑ String: 字符串类型。
- ❑ Integer: 整数类型。
- ❑ Double: 实数类型。
- ❑ Date: 日期类型。
- ❑ Currency: 与货币相关的类型。

9.4.1 两个控件内容的比较验证

如果 CompareValidator 控件同时指定了 ControlToValidate 和 ControlToCompare 属性的值，那

么该控件将验证被指定两个控件的内容是否满足指定关系。效果如图 9.7 所示。

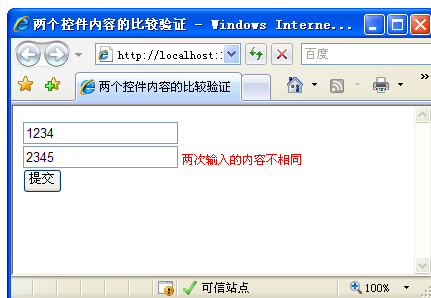


图 9.7 两个控件内容的比较验证

下面的代码使用了 CompareValidator 控件 cvValue 来验证 tbValue 和 tbCompareValue 控件的内容是否相等（由 Operator 属性默认指定）。如果这两个控件的内容不相等，则显示提示信息“两次输入的内容不相同”。另外，该示例还使用了 RequiredFieldValidator 验证控件 rfValue 来验证 tbValue 控件的内容是否为空。如果该内容为空，则 rfValue 控件显示提示信息“输入不能为空！”。

```
<!-- ASPNETValidator/Compare.aspx -->
<%@ Page Language="C#" AutoEventWireup="true" %>
<html xmlns="http://www.w3.org/1999/xhtml" >
<head runat="server"><title>两个控件内容的比较验证</title></head>
<body><form id="form1" runat="server">
<asp:TextBox ID="tbValue" runat="server"></asp:TextBox>
<asp:RequiredFieldValidator ID="rfValue" runat="server"
ErrorMessage="输入不能为空！" Font-Size="9pt" ControlToValidate="tbValue"
Display="Dynamic"></asp:RequiredFieldValidator>
<asp:TextBox ID="tbCompareValue" runat="server"></asp:TextBox>
<asp:CompareValidator ID="cvValue" runat="server"
ControlToCompare="tbValue" ControlToValidate="tbCompareValue"
ErrorMessage="两次输入的内容不相同"
Font-Size="9pt"></asp:CompareValidator>
<br />
<asp:Button ID="btnCheck" runat="server" Text="提交" />
</form></body>
</html>
```

9.4.2 检查控件内容的数据类型

如果 CompareValidator 控件指定了 ControlToValidate 属性的值，同时将 Operator 属性的值设置为“DataTypeCheck”。那么该控件将验证被验证控件的内容能否转换为指定的数据类型。效果如图 9.8 所示。

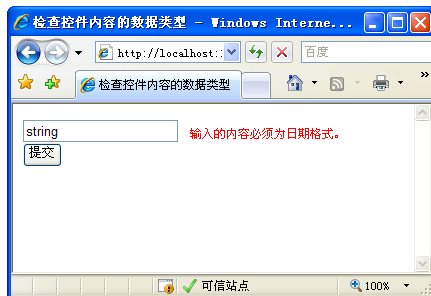


图 9.8 检查控件内容的数据类型

下面的代码实例使用了 CompareValidator 控件 cvValue 来验证 tbValue 控件的内容是否为日期类型。其中，日期类型由 Type 属性指定，满足的关系由 Operator 属性指定。如果控件的内容不能转换为日期类型，则验证失败，并显示提示信息“输入的内容必须为日期格式。”。另外，该示例还使用了 RequiredFieldValidator 验证控件 rfValue 来验证 tbValue 控件的内容是否为空。如果该内容为空，则 rfValue 控件显示提示信息“输入不能为空！”。

```
<!-- ASPNETValidator/CompareCheck.aspx -->
<%@ Page Language="C#" AutoEventWireup="true" %>
<html xmlns="http://www.w3.org/1999/xhtml" >
<head runat="server"><title>检查控件内容的数据类型</title></head>
<body><form id="form1" runat="server">
<asp:TextBox ID="tbValue" runat="server"></asp:TextBox>
<asp:RequiredFieldValidator ID="rfValue" runat="server"
ErrorMessage="输入不能为空！" Font-Size="9pt" ControlToValidate="tbValue"
Display="Dynamic"></asp:RequiredFieldValidator>
<asp:CompareValidator ID="cvValue" runat="server"
ControlToValidate="tbValue" ErrorMessage="输入的内容必须为日期格式。"
Font-Size="9pt" Display="Dynamic" Operator="DataTypeCheck"
Type="Date"></asp:CompareValidator>
<br />
<asp:Button ID="btnCheck" runat="server" Text="提交" />
</form></body>
</html>
```

9.4.3 指定的值和控件内容的比较验证

如果 CompareValidator 控件同时指定了 ControlToValidate 和 ValueToCompare 属性的值，那么该控件将验证被验证控件的内容和指定的值是否满足指定关系。效果如图 9.9 所示。

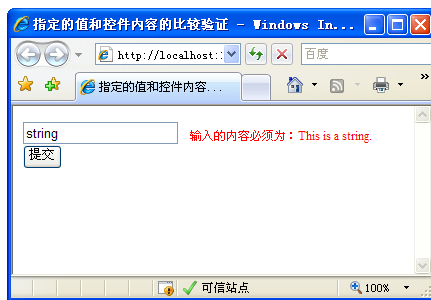


图 9.9 指定的值和控件内容的比较验证

下面的代码使用了 CompareValidator 控件 cvValue 来验证 tbValue 控件的内容和指定的值“`This is a string.`”是否相等。其中，比较的值由 ValueToCompare 属性指定。如果控件的内容和该值不相等，则显示提示信息“输入的内容必须为: `This is a string.`”。另外，该示例还使用了 RequiredFieldValidator 验证控件 rfValue 来验证 tbValue 控件的内容是否为空。如果该内容为空，则 rfValue 控件显示提示信息“输入不能为空！”。

```
<!-- ASPNETValidator/CompareValue.aspx -->
<%@ Page Language="C#" AutoEventWireup="true" %>
<html xmlns="http://www.w3.org/1999/xhtml" >
<head runat="server"><title>指定的值和控件内容的比较验证</title></head>
<body><form id="form1" runat="server">
<asp:TextBox ID="tbValue" runat="server"></asp:TextBox>
<asp:RequiredFieldValidator ID="rfValue" runat="server"
```

```

ErrorMessage="输入不能为空!" Font-Size="9pt" ControlToValidate="tbValue"
Display="Dynamic"></asp:RequiredFieldValidator>
<asp:CompareValidator ID="cvValue" runat="server"
ControlToValidate="tbValue"
ErrorMessage="输入的内容必须为: This is a string." Font-Size="9pt"
Display="Dynamic" ValueToCompare="This is a string.">
</asp:CompareValidator>
<br />
<asp:Button ID="btnCheck" runat="server" Text="提交" />
</form></body>
</html>

```

9.5 2 种自定义验证

CustomValidator 控件是最为灵活的验证控件，开发人员可以定义该控件的客户端函数或服务端函数，并在函数中执行需要的验证规则。CustomValidator 控件可以定义以下两种函数。

(1) 客户端函数。其中，控件的 ClientValidationFunction 属性指定客户端函数的名称，且该函数必须为以下形式。

```
function FunctionName(source,argument)
```

(2) 服务端函数。其中，控件的 OnServerValidate 属性指定服务端函数的名称，且该函数必须满足以下形式。

```
void FunctionName(object source,ServerValidateEventArgs args)
```

ServerValidateEventArgs 参数包含两个属性：IsValid 和 Value。其中，Value 属性表示被验证控件的内容；IsValid 属性表示验证的结果，如果通过验证，则它的值为 true，否则为 false。

9.5.1 自定义客户端验证

如果为 CustomValidator 控件定义了客户端函数，那么该控件可以在客户端对用户的输入执行验证。效果如图 9.10 所示。

注意：上述验证运算不需要提交到服务器，因此验证过程中，页面不会刷新。

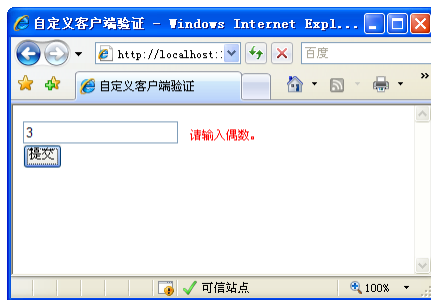


图 9.10 自定义客户端验证

下面的代码使用了 CustomValidator 控件 cvValue 来验证 tbValue 控件的内容是否为偶数，并定义了 cvValue 控件的客户端验证函数 EvenNumber(source,argument)。该函数判断用户输入的是否为偶数。如果上述验证失败，则显示提示信息“请输入偶数。”。另外，该示例还使用了 RequiredFieldValidator 验证控件 rfValue 来验证 tbValue 控件的内容是否为空。如果该内容为空，

则 rfValue 控件显示提示信息“输入不能为空!”。

```
<!-- ASPNETValidator/CustomServer.aspx -->
<%@ Page Language="C#" AutoEventWireup="true" %>
<html xmlns="http://www.w3.org/1999/xhtml" >
<head runat="server"><title>自定义客户端验证</title></head>
<body><form id="form1" runat="server">
<asp:TextBox ID="tbValue" runat="server"></asp:TextBox>
<asp:RequiredFieldValidator ID="rfValue" runat="server"
ErrorMessage="输入不能为空!" Font-Size="9pt" ControlToValidate="tbValue"
Display="Dynamic"></asp:RequiredFieldValidator>
<asp:CustomValidator ID="cvValue" runat="server"
ControlToValidate="tbValue" ErrorMessage="请输入偶数." Font-Size="9pt"
Display="Dynamic" ClientValidationFunction="EvenNumber" >
</asp:CustomValidator><br />
<asp:Button ID="btnCheck" runat="server" Text="提交" />
</form>
<script language="javascript" type="text/javascript">
function EvenNumber(source,argument)
{
/* 验证用户输入是否为偶数 */
if(argument.Value % 2 == 0)argument.IsValid = true;
else argument.IsValid = false;
}
</script>
</form></html>
```

9.5.2 自定义服务端验证

如果为 CustomValidator 控件定义了服务端函数,那么该控件可以在服务器端对用户的输入执行验证。效果如图 9.11 所示。

注意: 上述验证运算需要提交到服务器,因此在验证过程中,页面会刷新一次。

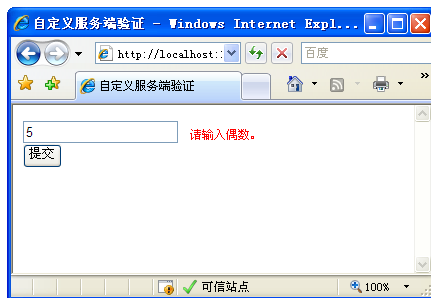


图 9.11 自定义服务端验证

下面的代码使用了 CustomValidator 控件 cvValue 来验证 tbValue 控件的内容是否为偶数,并定义了 cvValue 控件的服务端验证函数 cvValue_ServerValidate(object source, ServerValidateEventArgs args)。该函数首先判断用户输入是否为整数,如果是,则进一步验证是否为偶数。如果上述验证失败,则显示提示信息“请输入偶数。”。另外,该示例还使用 RequiredFieldValidator 验证控件 rfValue 来验证 tbValue 控件的内容是否为空。如果该内容为空,则 rfValue 控件显示提示信息“输入不能为空!”。

```
<!-- ASPNETValidator/CustomClient.aspx -->
<%@ Page Language="C#" %>
<script runat="server">
```

```
protected void cvValue_ServerValidate(object source,
ServerValidateEventArgs args)
{
    int value = -1;
    ///判断用户输入是否为整数
    if(Int32.TryParse(args.Value,out value) == false)
    {
        args.IsValid = false;return;
    }
    ///判断用户输入是否为偶数
    if(value % 2 == 0){args.IsValid = true;}
    else{args.IsValid = false;}
}
</script>
<html xmlns="http://www.w3.org/1999/xhtml" >
<head runat="server"><title>自定义服务端验证</title></head>
<body><form id="form1" runat="server">
<asp:TextBox ID="tbValue" runat="server"></asp:TextBox>
<asp:RequiredFieldValidator ID="rfValue" runat="server"
ErrorMessage="输入不能为空!" Font-Size="9pt" ControlToValidate="tbValue"
Display="Dynamic"></asp:RequiredFieldValidator>
<asp:CustomValidator ID="cvValue" runat="server"
ControlToValidate="tbValue" ErrorMessage="请输入偶数。" Font-Size="9pt"
Display="Dynamic"
OnServerValidate="cvValue_ServerValidate"></asp:CustomValidator><br />
<br />
<asp:Button ID="btnCheck" runat="server" Text="提交" />
</form></body>
</html>
```

9.6 7 种正则表达式验证

RegularExpressionValidator 控件可以验证控件的内容是否满足指定的正则表达式。其中，正则表达式由 ValidationExpression 属性指定。因此，该控件可以根据不同的正则表达式来形成不同的验证方式。

9.6.1 整数验证

如果 RegularExpressionValidator 控件指定的正则表达式能够验证整数，那么该控件可以验证用户的输入是否为整数。效果如图 9.12 所示。

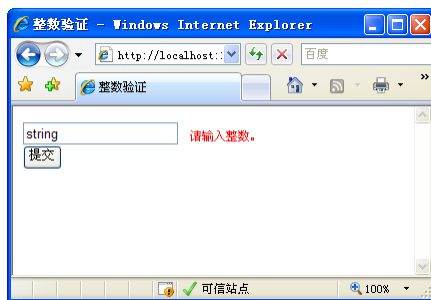


图 9.12 整数验证

下面的代码使用了 `RegularExpressionValidator` 控件 `revValue` 来验证 `tbValue` 控件的内容是否为整数。如果该内容不是整数，则验证失败，并显示提示信息“请输入整数。”。另外，该示例还使用了 `RequiredFieldValidator` 验证控件 `rfValue` 验证 `tbValue` 控件的内容是否为空。如果该内容为空，则 `rfValue` 控件显示提示信息“输入不能为空！”。

正则表达式 `^(0|-?[1-9]\d*)$` 用来验证一个整数，包括正整数、零和负整数。该表达式的说明如下。

- ❑ 0 验证整数零。
- ❑ `[1-9]\d*` 验证以非零开头的任何正整数。
- ❑ `-?` 可以验证符号“-”，该符号可以出现一次或者不出现。当它不出现时，此时被验证的内容为正整数，否则为负整数。

```
<!-- ASPNETValidator/REInt.aspx -->
<%@ Page Language="C#" AutoEventWireup="true" %>
<html xmlns="http://www.w3.org/1999/xhtml" >
<head runat="server"><title>整数验证</title></head>
<body><form id="form1" runat="server">
<asp:TextBox ID="tbValue" runat="server"></asp:TextBox>
<asp:RequiredFieldValidator ID="rfValue" runat="server"
ErrorMessage="输入不能为空!" Font-Size="9pt" ControlToValidate="tbValue"
Display="Dynamic"></asp:RequiredFieldValidator>
<asp:RegularExpressionValidator ID="revValue" runat="server"
ControlToValidate="tbValue" ErrorMessage="请输入整数。" Font-Size="9pt"
ValidationExpression="^(0|-?[1-9]\d*)$" >
</asp:RegularExpressionValidator>
<br />
<asp:Button ID="btnCheck" runat="server" Text="提交" />
</form></body>
</html>
```

9.6.2 数值验证

如果 `RegularExpressionValidator` 控件指定的正则表达式能够验证数值，那么该控件可以验证用户的输入是否为一个数值。效果如图 9.13 所示。



图 9.13 数值验证

下面的代码使用了 `RegularExpressionValidator` 控件 `revValue` 来验证 `tbValue` 控件的内容是否为数值。如果该内容不是数值，则验证失败，并显示提示信息“请输入整数或实数。”。另外，该示例还使用了 `RequiredFieldValidator` 验证控件 `rfValue` 来验证 `tbValue` 控件的内容不能为空。如果该内容为空，则 `rfValue` 控件显示提示信息“输入不能为空！”。

正则表达式 `-(0|([1-9]\d+))(\.\d+)?` 用来验证一个整数或实数，包括正整数、正实数、零、负整

数和负实数。该表达式的说明如下。

- ❑ 0 验证整数零。
- ❑ [1-9]\d*验证以非零开头的任何正整数。
- ❑ -?可以验证符号“-”，该符号可以出现一次或者不出现。当它不出现时，此时被验证的内容为正整数或正实数，否则为负整数或负实数。
- ❑ \. \d+验证从小数点开始的内容（包括小数点）。
- ❑ (\. \d+)?表示小数点开始的内容可以出现一次或者不出现。如果出现，则验证实数，否则为整数。

```
<!-- ASPNETValidator/RENumber.aspx -->
<%@ Page Language="C#" AutoEventWireup="true" %>
<html xmlns="http://www.w3.org/1999/xhtml" >
<head runat="server"><title>数值验证</title></head>
<body><form id="form1" runat="server">
<asp:TextBox ID="tbValue" runat="server"></asp:TextBox>
<asp:RequiredFieldValidator ID="rfValue" runat="server"
ErrorMessage="输入不能为空！" Font-Size="9pt" ControlToValidate="tbValue"
Display="Dynamic"></asp:RequiredFieldValidator>
<asp:RegularExpressionValidator ID="revValue" runat="server"
ControlToValidate="tbValue" ErrorMessage="请输入整数或实数。"
Font-Size="9pt"
ValidationExpression="-?(0|([1-9]\d*))(\. \d+)?">
</asp:RegularExpressionValidator>
<br />
<asp:Button ID="btnCheck" runat="server" Text="提交" />
</form></body>
</html>
```

9.6.3 电话号码验证

如果 RegularExpressionValidator 控件指定的正则表达式能够验证电话号码，那么该控件可以验证用户的输入是否为一个电话号码。效果如图 9.14 所示。

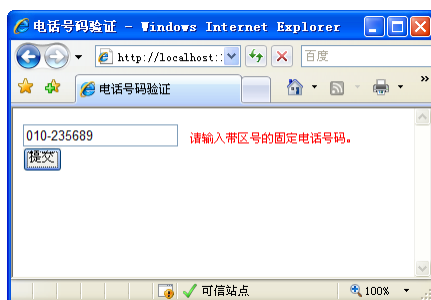


图 9.14 电话号码验证

下面的代码使用了 RegularExpressionValidator 控件 revValue 来验证 tbValue 控件的内容是否为固定电话号码。如果该内容不是固定电话号码，则验证失败，并显示提示信息“请输入带区号的固定电话号码。”。另外，该示例还使用了 RequiredFieldValidator 验证控件 rfValue 来验证 tbValue 控件的内容是否为空。如果该内容为空，则 rfValue 控件显示提示信息“输入不能为空！”。

正则表达式 `(\d{3,4})\d{3,4}-?\d{7,8}` 用来验证输入的内容是否为固定电话号码。该表达式的说明如下。

- \d{3,4} 验证 3 到 4 位的区号。
- (\d{3,4}) 验证 3 到 4 位的区号，且被小括号包围。
- \d{3,4}- 验证 3 到 4 位的区号，且后接一个连接字符“-”。
- \d{7,8} 验证 7 到 8 位的电话号码。
- ((\d{3,4})|\d{3,4}-)? 表示被验证的电话号码可以添加区号，或者不添加。

```
<!-- ASPNETValidator/REPhone.aspx -->
<%@ Page Language="C#" AutoEventWireup="true" %>
<html xmlns="http://www.w3.org/1999/xhtml" >
<head runat="server"><title>电话号码验证</title></head>
<body><form id="form1" runat="server">
<asp:TextBox ID="tbValue" runat="server"></asp:TextBox>
<asp:RequiredFieldValidator ID="rfValue" runat="server"
ErrorMessage="输入不能为空！" Font-Size="9pt" ControlToValidate="tbValue"
Display="Dynamic"></asp:RequiredFieldValidator>
<asp:RegularExpressionValidator ID="revValue" runat="server"
ControlToValidate="tbValue" ErrorMessage="请输入带区号的固定电话号码。"
Font-Size="9pt"
ValidationExpression="((\d{3,4})|\d{3,4}-)?\d{7,8}">
</asp:RegularExpressionValidator>
<br />
<asp:Button ID="btnCheck" runat="server" Text="提交" />
</form></body>
</html>
```

9.6.4 身份证号码验证

如果 RegularExpressionValidator 控件指定的正则表达式能够验证身份证号码，那么该控件可以验证用户输入的是不是一个身份证号码。效果如图 9.15 所示。



图 9.15 身份证号码验证

下面的代码使用了 RegularExpressionValidator 控件 revValue 来验证 tbValue 控件的内容是否为身份证号码。如果该内容不是身份证号码，则验证失败，并显示提示信息“请输入 15 位或 18 位身份证号码。”。另外，该示例还使用了 RequiredFieldValidator 验证控件 rfValue 来验证 tbValue 控件的内容是否为空。如果该内容为空，则 rfValue 控件显示提示信息“输入不能为空！”。

正则表达式(\d{17}(\d|xX))\d{15}用来验证输入的内容是否为身份证号码。该表达式的说明如下。

- \d{15} 验证 15 位的身份证号码。
- \d{17}(\d|xX) 验证 18 位的身份证号码。其中，最后一个号码可以是数字或字母 X。

```
<!-- ASPNETValidator/REIdentity.aspx -->
<%@ Page Language="C#" AutoEventWireup="true" %>
```

```
<html xmlns="http://www.w3.org/1999/xhtml" >
<head runat="server"><title>身份证号码验证</title></head>
<body><form id="form1" runat="server">
<asp:TextBox ID="tbValue" runat="server"></asp:TextBox>
<asp:RequiredFieldValidator ID="rfValue" runat="server"
ErrorMessage="输入不能为空！" Font-Size="9pt" ControlToValidate="tbValue"
Display="Dynamic"></asp:RequiredFieldValidator>
<asp:RegularExpressionValidator ID="revValue" runat="server"
ControlToValidate="tbValue" ErrorMessage="请输入 15 位或 18 位身份证号码。"
Font-Size="9pt" ValidationExpression="(\\d{17}(\\d|x|X))|\\d{15}">
</asp:RegularExpressionValidator>
<br />
<asp:Button ID="btnCheck" runat="server" Text="提交" />
</form></body>
</html>
```

9.6.5 电子邮件验证

如果 `RegularExpressionValidator` 控件指定的正则表达式能够验证电子邮件，那么该控件可以验证用户的输入是否为一个电子邮件。效果如图 9.16 所示。

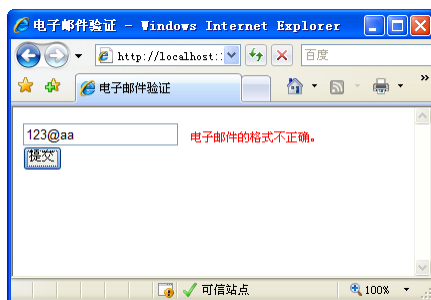


图 9.16 电子邮件验证

下面的代码使用了 `RegularExpressionValidator` 控件 `revValue` 来验证 `tbValue` 控件的内容是否为电子邮件。如果该内容不是电子邮件，则验证失败，并显示提示信息“电子邮件的格式不正确”。另外，该示例还使用了 `RequiredFieldValidator` 验证控件 `rfValue` 来验证 `tbValue` 控件的内容是否为空。如果该内容为空，则 `rfValue` 控件显示提示信息“输入不能为空！”。

正则表达式 `\w+([-+.]w+)*@w+([-.]w+)*\w+([-.]w+)*` 用来验证输入的内容是否为电子邮件。该表达式的说明如下。

- `\w+`能够匹配长度至少为 1、由单词字符组成的字符串。
- `[-+.]`匹配-、+、.、'字符；`[-+.]w+`匹配以-、+、.或'字符开头的、后接长度至少为 1 的单词字符串；`([-+.]w+)*`表示以-、+、.或'字符开头的、后接长度至少为 1 的单词字符串可以不出现或者至少出现 1 次。
- `@`匹配邮件地址中的字符@。
- `[-.]`匹配-、.字符；`[-.]w+`匹配以-或.字符开头的、后接长度至少为 1 的单词字符串；`([-.]w+)*`表示以-或.字符开头的、后接长度至少为 1 的单词字符串可以不出现或者至少出现 1 次。
- `\.`匹配字符。

```
<!-- ASPNETValidator/REEmail.aspx -->
<%@ Page Language="C#" AutoEventWireup="true" %>
```

```
<html xmlns="http://www.w3.org/1999/xhtml" >
<head runat="server"><title>电子邮件验证</title></head>
<body><form id="form1" runat="server">
<asp:TextBox ID="tbValue" runat="server"></asp:TextBox>
<asp:RequiredFieldValidator ID="rfValue" runat="server"
ErrorMessage="输入不能为空！" Font-Size="9pt" ControlToValidate="tbValue"
Display="Dynamic"></asp:RequiredFieldValidator>
<asp:RegularExpressionValidator ID="revValue" runat="server"
ControlToValidate="tbValue" ErrorMessage="电子邮件的格式不正确。"
Font-Size="9pt"
ValidationExpression="\w+([-+.'\w+)*@\w+([-.\w+)*\.\w+([-.\w+)*">
</asp:RegularExpressionValidator>
<br />
<asp:Button ID="btnCheck" runat="server" Text="提交" />
</form></body>
</html>
```

9.6.6 日期和时间验证

如果 `RegularExpressionValidator` 控件指定的正则表达式能够验证日期和时间，那么该控件可以验证用户输入的是不是一个日期和时间。效果如图 9.17 所示。

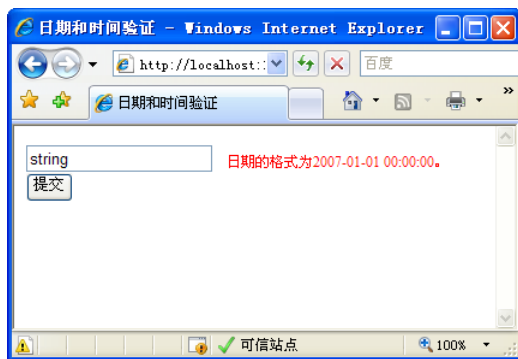


图 9.17 日期和时间验证

下面的代码使用了 `RegularExpressionValidator` 控件 `revValue` 来验证 `tbValue` 控件的内容是否为日期（包含时间部分）。如果该内容不是日期（包含时间部分），则验证失败，并显示提示信息“日期的格式为 2007-01-01 00:00:00。”。另外，该示例还使用了 `RequiredFieldValidator` 验证控件 `rfValue` 来验证 `tbValue` 控件的内容不能为空。如果该内容为空，则 `rfValue` 控件显示提示信息“输入不能为空！”。

正则表达式 `^d{4}(?<connectchar>[-/])((1[0-2])|(0?\d))k<connectchar>((1[2]\d)|(3[01])|(0?\d))((1[0-1]\d)|(2[0-3]))(:[0-5]\d){2}$` 用来验证输入的内容是否为日期（包含时间部分）。该表达式的说明如下。

- `d{4}` 验证 4 位的年字符串。
- `(?<connectchar>[-/])` 验证年月日之间的连接字符，并保存为名为“connectchar”的分组。
- `(1[0-2])|(0?\d)` 验证 1 或 2 位的月字符串。
- `k<connectchar>` 后向引用年月之间的连接字符。
- `(1[2]\d)|(3[01])|(0?\d)` 验证 1 或 2 位的日字符串。
- `((1[0-1]\d)|(2[0-3]))` 验证时间部分的小时字符串。

- `(:[0-5]\d){2}` 验证时间部分的分钟和秒，并包含连接字符“:”。

```
<!-- ASPNETValidator/REDate.aspx -->
<%@ Page Language="C#" AutoEventWireup="true" %>
<html xmlns="http://www.w3.org/1999/xhtml" >
<head runat="server"><title>日期和时间验证</title></head>
<body><form id="form1" runat="server">
<asp:TextBox ID="tbValue" runat="server"></asp:TextBox>
<asp:RequiredFieldValidator ID="rfValue" runat="server"
ErrorMessage="输入不能为空！" Font-Size="9pt" ControlToValidate="tbValue"
Display="Dynamic"></asp:RequiredFieldValidator>
<asp:RegularExpressionValidator ID="revValue" runat="server"
ControlToValidate="tbValue"
ErrorMessage="日期的格式为 2007-01-01 00:00:00。" Font-Size="9pt"
ValidationExpression="^\\d{4}(?<connectchar>[-/\\. ])(1[0-2])|(0?\\d))\\k<connect
char>((1[2]\\d)|(3[01])|(0?\\d)) (([0-1]\\d)|(2[0-3]))(:[0-5]\\d){2}$">
</asp:RegularExpressionValidator>
<br />
<asp:Button ID="btnCheck" runat="server" Text="提交" />
</form></body>
</html>
```

9.6.7 URL 验证

如果 `RegularExpressionValidator` 控件指定的正则表达式能够验证 URL，那么该控件可以验证用户的输入是否为一个 URL（或 HTTP 地址）。效果如图 9.18 所示。

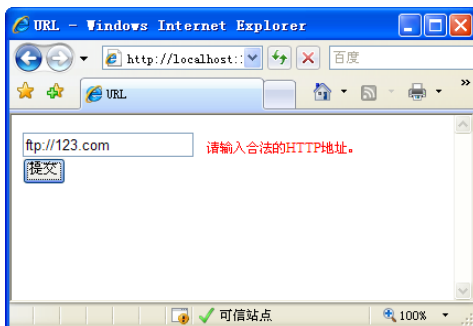


图 9.18 URL 验证

下面的代码使用了 `RegularExpressionValidator` 控件 `revValue` 来验证 `tbValue` 控件的内容是否为 URL（或 HTTP 地址）。如果该内容不是 URL（或 HTTP 地址），则验证失败，并显示提示信息“请输入合法的 HTTP 地址。”。另外，该示例还使用了 `RequiredFieldValidator` 验证控件 `rfValue` 来验证 `tbValue` 控件的内容是否为空。如果该内容为空，则 `rfValue` 控件显示提示信息“输入不能为空！”。

正则表达式 `http(s)?://([w-]+\.)+[w-]+(/[w- ./?%&=]*)?` 用来验证输入的内容是否为 URL（或 HTTP 地址）。该表达式的说明如下。

- `[w-]` 能够匹配单词字符和连接符号-。
- `\.` 匹配字符。
- `[w-]+\.` 能够匹配由单词字符和连接符号-组成的以字符串开头的、以字符.结尾的字符串。
- `([w-]+\.)+` 能够匹配 1 个或多个由单词字符和连接符号-组成的、以字符串开头的、以字符.结尾的字符串。

- /匹配字符/。
- `[w- ./?%&=]`能够匹配单词字符、-、（空格）、.、/、?、%、&和=；`[w- ./?%&=]*`能够匹配空字符串，或者由单词字符、-、（空格）、.、/、?、%、&和=组成的长度至少为1的字符串。
- `([w- ./?%&=]*)?`表示表达式`[w- ./?%&=]*`匹配的字符串可以出现或者出现1次。

```
<!-- ASPNETValidator/REUrl.aspx -->
<%@ Page Language="C#" AutoEventWireup="true" %>
<html xmlns="http://www.w3.org/1999/xhtml" >
<head runat="server"><title>URL</title></head>
<body><form id="form1" runat="server">
<asp:TextBox ID="tbValue" runat="server"></asp:TextBox>
<asp:RequiredFieldValidator ID="rfValue" runat="server"
ErrorMessage="输入不能为空！" Font-Size="9pt" ControlToValidate="tbValue"
Display="Dynamic"></asp:RequiredFieldValidator>
<asp:RegularExpressionValidator ID="revValue" runat="server"
ControlToValidate="tbValue" ErrorMessage="请输入合法的 HTTP 地址。"
Font-Size="9pt"
ValidationExpression="http(s)?://([w-]+\.)+[w-]+(/[w- ./?%&=]*)?">
</asp:RegularExpressionValidator>
<br />
<asp:Button ID="btnCheck" runat="server" Text="提交" />
</form></body>
</html>
```

9.7 2 种显示验证摘要

ValidationSummary 控件不是一个验证控件，但是它可以用来处理或集中显示页面上所有验证控件的提示或错误信息。另外，页面上所有验证控件的提示或错误信息又称为验证摘要。该控件可以用两种方式显示验证摘要：在网页上显示和在弹出对话框上显示。其中，显示方式由 ShowMessageBox 属性指定。如果 ShowMessageBox 属性的值为 true，则在弹出对话框上显示，否则在网页上显示。

ValidationSummary 控件的 DisplayMode 属性指定了多条信息的显示模式，它的值的具体说明如下。

- BulletList，以项目符号列表形式显示验证摘要，为系统默认值。
- List，以列表形式显示验证摘要。
- SingleParagraph，以单个段落形式显示验证摘要。

注意：若需要使用 ValidationSummary 控件显示页面上所有验证控件的提示或错误信息，那么必须把页面上的验证控件的 Display 属性的值设置为“None”。

9.7.1 在对话框上显示验证摘要

如果把网页上所有验证控件的 Display 属性的值都设置为“None”，并把 ValidationSummary 控件的 ShowMessageBox 属性的值设置为 true，那么该控件将在弹出对话框中显示验证摘要。效果如图 9.19 所示。

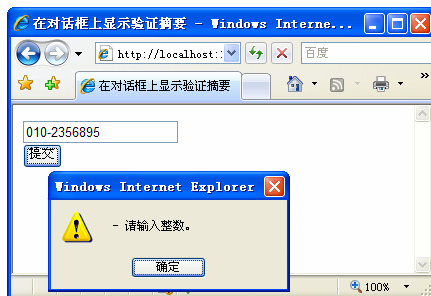


图 9.19 在对话框上显示验证摘要

下面的代码使用了 `ValidationSummary` 控件 `vsMessage` 在弹出对话框中显示网页（`SummaryPage.aspx`）中的所有验证控件（`rfValue` 和 `revValue` 控件）的提示或错误信息。另外，该示例还使用了 `RequiredFieldValidator` 验证控件 `rfValue` 来验证 `tbValue` 控件的内容不能为空。如果该内容为空，则 `rfValue` 控件显示提示信息“输入不能为空！”。

```
<!-- ASPNETValidator/SummaryDialog.aspx -->
<%@ Page Language="C#" %>
<html xmlns="http://www.w3.org/1999/xhtml" >
<head runat="server"><title>在对话框上显示验证摘要</title></head>
<body><form id="form1" runat="server">
<asp:TextBox ID="tbValue" runat="server"></asp:TextBox>
<asp:RequiredFieldValidator ID="rfValue" runat="server"
ErrorMessage="输入不能为空！" Font-Size="9pt" Display="None"
ControlToValidate="tbValue"></asp:RequiredFieldValidator>
<asp:RegularExpressionValidator ID="revValue" runat="server"
ControlToValidate="tbValue" ErrorMessage="请输入整数。" Font-Size="9pt"
ValidationExpression="^(0|-?[1-9]\d*)$"
Display="None"></asp:RegularExpressionValidator>
<br /><asp:Button ID="btnCheck" runat="server" Text="提交" />
<br /><asp:ValidationSummary ID="vsMessage" runat="server"
Font-Size="9pt" ShowMessageBox="True" />
</form></body>
</html>
```

9.7.2 在网页上显示验证摘要

如果把网页上所有验证控件的 `Display` 属性的值都设置为“None”，那么 `ValidationSummary` 控件将默认在网页上显示验证摘要。效果如图 9.20 所示。

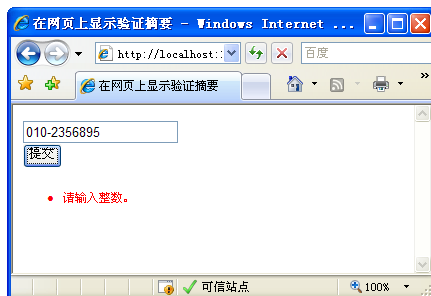


图 9.20 在网页上显示验证摘要

下面的代码使用了 `ValidationSummary` 控件 `vsMessage` 在网页上直接显示网页（`SummaryPage.aspx`）

中的所有验证控件（rfValue 和 revValue 控件）的提示或错误信息。另外，该实例还使用了 RequiredFieldValidator 验证控件 rfValue 来验证 tbValue 控件的内容不能为空。如果该内容为空，则 rfValue 控件显示提示信息“输入不能为空！”。

```
<!-- ASPNETValidator/SummaryPage.aspx -->
<%@ Page Language="C#" %>
<html xmlns="http://www.w3.org/1999/xhtml" >
<head runat="server"><title>在网页上显示验证摘要</title></head>
<body><form id="form1" runat="server">
<asp:TextBox ID="tbValue" runat="server"></asp:TextBox>
<asp:RequiredFieldValidatorID="rfValue" runat="server"
ErrorMessage="输入不能为空！" Font-Size="9pt" Display="None"
ControlToValidate="tbValue"></asp:RequiredFieldValidator>
<asp:RegularExpressionValidator ID="revValue" runat="server"
ControlToValidate="tbValue" ErrorMessage="请输入整数。" Font-Size="9pt"
ValidationExpression="^(0|-?[1-9]\d*)$"
Display="None"></asp:RegularExpressionValidator>
<br /><asp:Button ID="btnCheck" runat="server" Text="提交" />
<br />
<asp:ValidationSummary ID="vsMessage" runat="server" Font-Size="9pt" />
</form></body>
</html>
```

第 3 篇 PHP 正则表达式应用

第 10 章 常见 PHP 数据类型

PHP 支持以下数据类型：布尔型、NULL 型、整型、浮点型、字符串型，以及数组、对象和资源型。这些数据类型之间可以进行转化，这给 PHP 的使用带来极大的方便。本章将详细介绍这些常见的数据类型以及它们之间的相互转化。

10.1 7 种 PHP 常见数据类型

与其他程序语言相比，PHP 是一种轻量级的程序开发语言，其数据类型也比较简单。PHP 中的常见数据类型主要有以下几种。

- Boolean，布尔型
- NULL 型
- Integer，整型
- Float/Double，浮点型
- String，字符串型
- Array，数组
- Object，对象
- Resource，资源型

其中，布尔型、NULL 型、整型、浮点型和字符串型是基本的数据类型。它们中只能存储一个数据，是数据结构中最基本的单元。数组、对象和资源则属于复合型，它们通常由一个或多个基本的数据类型组合而成。

10.1.1 布尔型

布尔型变量只有两个有效的值 True 或 False，即“真”或“假”。与布尔型变量等价的整型变量分别为 1 和 0。

10.1.2 NULL 型

NULL 值表示“空”、“没有”、“无效的”、“不存在”或者“未被初始化”等概念。它只有一个值，就是关键字 NULL。

下列情况下的变量被认为是 NULL。

- 没有被赋值或者初始化。
- 被赋值为 NULL。
- 被清空，如使用了函数 `unset()`。

10.1.3 整型

整型变量表示整数。PHP 不支持无符号整数。整数通常使用十进制数表示，也可以用八进制数或十六进制数表示。用八进制数表示时，前面要加上标识符“0”（数字零）；使用十六进制数表示时，要加上“0x”。例如：

- 八进制数 0267，转换成十进制数为 $2 \times 8^2 + 6 \times 8 + 7 = 183$ ；
- 十六进制数 0xEF9A，转换成十进制数为 $14 \times 16^3 + 15 \times 16^2 + 9 \times 16 + 10 = 61338$ 。

10.1.4 浮点型

浮点数通常指实数，PHP 中只能用十进制数表示。浮点数只是一种近似的数值，浮点寄存器通常不能精确地存储浮点数，从而产生误差。因此，在对其进行运算和比较时应特别注意。例如，下面两段代码返回的结果并不一样。

```
<?php
$a1 = 5.03e+21 + 0.01 - 5.03e+21;
echo $a1;           //结果是 0

$a2 = 5.03e+21 - 5.03e+21 + 0.01;
echo $a2;           //结果是 0.01
?>
```

10.1.5 字符串

字符串是一系列字符的组合。通常使用一对单引号“'”或双引号“”来定义字符串。

1. 转义字符

在字符串中通常需要输出一些特殊字符，它们可以通过转义字符实现。这些特殊字符通常是一些非可打印字符，表 10.1 列出了 PHP 中常用的转义字符。

表 10.1 PHP 中的转义字符

转义字符	含义
\n	换行符（LF）
\r	回车符（CR）
\t	水平制表符（HT）
\\	反斜线
\\$	美元符号
\"	双引号
\NNN	用八进制表示的字符（N 代表一个 0~7 的数字）
\xNN	用十六进制表示的字符（N 表示一个 0~9、A~F 的字符）

如果要在双引号中包含双引号“”、反斜线“\”或者“\$”等字符，就必须在它们前面加一个“\”。而在单引号字符串中包含“'”时，也要在前面加一个“\”进行转义。对于非转义字符，PHP 不做任何解释，只是原样重复。下面给出了一些简单的代码片断。

```
<?php
$tom = "汤姆";
$jerry = "杰瑞";

$lovemessage1 = "$tom 与 $jerry! \n$jerry 与 $tom!\n"; //这里使用双引号
$lovemessage2 = '$tom 与 $jerry! \n$jerry 与 $tom!\n'; //这里使用单引号
```

```
echo $lovemessage1;
echo $lovemessage2;
?>
```

以下是输出的结果。前两行是\$lovemessage1 的输出，第三行是\$lovemessage2 的输出。

```
汤姆 与 杰瑞!
杰瑞 与 汤姆!
$tom 与 $jerry! \n$jerry 与 $tom!\n
```

显然，使用双引号时，程序对字符串中的变量进行了解析。PHP 解析器会尽可能多地与“\$”后面的字符匹配，形成一个合法的变量名，除非遇到标点符号或者空白。为了更明确地表明一个变量，可以用花括号把变量名括起来。将上面的代码改写一下，会得到不同的效果。

```
<?php
    $tom = "汤姆";
    $jerry = "杰瑞";

    $lovemessage3 = "$tom与$jerry! \n$jerry与$tom!\n"; //去掉了不必要空格
    $lovemessage4 = "${tom}与${jerry}! \n${jerry}与${tom}!\n"; //使用了{}确定变量
    echo $lovemessage3;
    echo $lovemessage4;
?>
```

以下是输出的结果。前两行是\$lovemessage3 的输出，后两行是\$lovemessage4 的输出。

```
杰瑞!
汤姆!
汤姆与杰瑞!
杰瑞与汤姆!
```

由于 PHP 将中文也作为合法的变量名，因此\$lovemessage3 中匹配了更长的变量名“\$tom 与”，而这个变量并不存在。\$lovemessage4 则使用了“{}”对变量名进行明确的界定，因此输出了正确的结果。需要指出的是，“{ \$tom}”和“\${me}”的两种书写方式都是正确的。

2. 定界标识符

此外，PHP 中还引入了另一种方便的字符串定义方法。使用定界符“<<<”来定义字符串，字符串必须包含在一组定界标识符内。

定界符“<<<”的后面紧接着的是定界标识符。定界标识符由字母、数字或下画线构成，并且不能以数字开始。结尾的标识符必须顶格书写，并且其前面不能有任何其他字符。最典型的错误是将结尾的定界标识符进行缩进。

定界符中的字符串可以被解析，可以使用转义字符，但不必转义引号（“'”和“””）。当需要定义或输出大量的文本时，这是尤为方便的。下面的代码演示了定界符的使用。

```
<?php
// 使用定界符定义一个字符串
$string = <<<EOT_STR
    Example of string spanning multiple lines,
    $this_is_value,
    <font color="#00FFFF">HTML</font>
EOT_STR; //定界结束符必须顶格书写

echo $string;
?>
```

10.1.6 数组

数组 (Array) 是 PHP 中的一种重要的数据类型。数组中可以存放多种数据类型：布尔值、数字、字符串、数组、对象和资源等。

数组中的每一个元素都由索引 (键名) 和值两部分组成。“索引”和“键名”指的是同一样东西，它只能由数字或字符串构成。索引是数字的数组，与 C 语言中的标准数组类似，也被称为“数字索引数组”。键名是字符串的数组，通常被称为“关联数组”或“结合数组”。

在 PHP 中，可以使用函数 `array()`，以及分别对元素赋值的方法构造数组。

1. 数组构造函数 `array()`

严格地说，`array()` 是一个 PHP 专门的语言结构。它接受指定数目的用逗号分隔的“key => value”参数对。key 和 value 作为索引和元素值，都可以用变量或常量表达，例如，下面的代码。

```
<?php
$key = 9;
$array_stuff = array(
    1,                                //0=>1
    'abc',                            //1=>'abc'
    13 => 'a bad number',             //以数字为键名
    "a sub array" => array(
        1, 2.5, 'x'
    ),                               //可以包含另一个数组
    $key => 'the dynamic key',        //键名是可以变化的
    'what is the key here?',         //14=> 'what is the key here?'
);

print_r($array_stuff);              //打印数组结构
?>
```

使用 `array()` 定义数组时，索引是可以省略的。省略的索引值将按当前下标最大索引值加 1 计算。上面代码中数组的最后一个元素之前的最大下标值是 13，那么该元素的索引就是 14。下面是这段代码运行的结果，它显示了数组的结构。

```
Array
(
    [0] => 1
    [1] => abc
    [13] => a bad number
    [a sub array] => Array
        (
            [0] => 1
            [1] => 2.5
            [2] => x
        )
    [9] => the dynamic key
    [14] => what is the key here?
)
```

2. 分别对元素赋值

可以分别对元素赋值。当有同名元素时，只保留最后一次的定义。也可以省略元素的索引，计算被省略的索引值的方法与前面一致。例如：

```
<?php
$food[] = 'Pear';                // food[0] = 'Pear'
$food[] = 'Apple';               // food[1] = 'Apple'
$food[5] = 'Banana';             // food[5] = 'Banana'
```

```
$food[] = 'Orange';           // food[6] = 'Orange'
?>
```

10.1.7 对象

对象是一种高级的数据类型。对象通常使用一个类进行描述。类（Class）由一组属性值和方法函数构成。属性表明对象的一种状态；方法则用来存取其属性值，或者进行某种相关运算。

PHP 中主要使用下面的关键字定义对象。

- **class**: 定义一个对象。
- **var**: 定义属性（PHP5 中还支持 **private**、**protect** 以及 **public** 等关键字）。
- **function**: 定义方法函数。

例如，下面是对汽车对象类的定义。对于一辆具体的汽车而言，它会具有一个名字或品牌（brand），在行驶中，它还具有特定的速度（velocity）。这个汽车对象 Car 也具有这些属性，同时，它还可以控制速度，进行行驶速度设定（setVelocity）、加速（addVelocity）或减速（reduceVelocity）等运算。

```
<?php
class Car {
    var $brand;      //汽车名称
    var $velocity = 0; //车速

    //构造器，创建一个汽车的实例
    function Car($brand){
        $this->brand = $brand;
    }

    //设置汽车的速度
    function setVelocity($velocity){
        $this->velocity = $velocity;
    }

    //加速运算
    function addVelocity($v){
        $this->velocity += $v;
    }

    //减速运算
    function reduceVelocity($v){
        $this->velocity -= $v;
    }
}

//以下是对 Car 类的测试
$car = new Car("Mazda");           //创建一个 Car 对象，名字为 Mazda。
$car->setVelocity(10);              //设置初始速度为 10
$car->addVelocity(2);               //加速：将当前速度加 2
$car->reduceVelocity(3);            //减速：将当前速度减 3

echo $car->velocity;                //当前速度为 9
?>
```

这里，仅就对象的概念做了简要的说明。要获得更多详细的介绍，可以参考后面有关“类与对象”的详细讨论。

10.2 5 种常见的类型转化

PHP 是一种弱类型程序语言，它不需要在声明变量时就确定变量的类型。一个变量的类型主要由它被实际使用时的上下文决定。为了明确这种不确定性，以及满足各种程序接口的需求，通常需要对变量进行类型识别和转化。

10.2.1 变量类型变化

PHP 不需要在声明变量时就确定变量的类型。变量的类型是由它被使用时的上下文决定。也就是说，如果把一个字符串值赋给变量 `$value`，则 `$value` 就是一个字符型变量；如果把一个整数赋给 `$value`，则它就变成了一个整型变量。

1. 类型变化

在表达式运算时，变量的类型经常自动变化。例如，当运算数之一是浮点数时，其他的运算数都会作为浮点型变量处理，而不管其原始类型是什么。其计算结果也将是浮点数。否则，运算数会作为整数处理。

这实际上是一种隐式的类型变化，它为 PHP 语言的使用增添了极大的灵活性。这种情形在实际开发中随处可见。下面的代码演示了这种变化情况。

```
<?php
    $value = "1";           // $value 是一个字符串，值是“1” (ASCII=49)
    $value ++;             // $value 是一个字符串，值是“2” (ASCII=50)
    $value += 50;          // $value 是一个整数 (52)
    $value = $value - 24.3; // $value 是一个浮点数 (25.7)
    $value = 15.1 - "10 people"; // $value 是一个浮点数 (5.1)
    $value = 5 + "10 people"; // $value 是一个整数 (15)
?>
```

2. 类型识别

一个特定的变量类型不是在任何时间都非常明显的，因为 PHP 能自己决定变量的类型。为了明确这种不确定性，首先需要对变量进行类型识别。PHP 中有一些函数，用于找出或判断某个变量的类型，这些函数如表 10.2 所示。

表 10.2 判断变量类型的函数

函数	返回值	含义
<code>gettype()</code>	字符串	取得变量的当前类型，返回如“boolean”、“integer”、“double”（由于历史原因，如果是浮点型数据，则返回“double”，而不是“float”）、“string”、“array”、“object”、“resource”或“NULL”等
<code>var_dump()</code>	字符串	变量调试，返回变量的值和数据类型
<code>is_bool()</code>	布尔值	判断变量是否为布尔型
<code>is_null()</code>	布尔值	判断变量是否为 NULL
<code>is_int()</code> 、 <code>is_integer()</code> 或 <code>is_long</code>	布尔值	判断变量是否为整型
<code>is_float()</code> 或 <code>is_real()</code>	布尔值	判断变量是否为浮点型
<code>is_numeric()</code>	布尔值	判断变量是否是数字或数字字符串
<code>is_nan()</code>	布尔值	判断变量是否为合法的数字类型
<code>is_array()</code>	布尔值	判断变量是否为数组型
<code>is_string()</code>	布尔值	判断变量是否为字符串型

续表

函数	返回值	含义
is_scalar()	布尔值	判断变量是否为标量类型，包括布尔型、NULL、整型、浮点型、数组型或者字符串型
is_object()	布尔值	判断变量是否为对象类型
is_resource()	布尔值	判断变量是否为资源类型

函数 `gettype()` 和 `var_dump()` 可以用于变量的查看和调试。它们都可以直观地反映变量的类型。而以 “`is_`” 开头的函数通常用于特定类型变量的判断。例如下面的代码片断。

```
<?php
    var_dump(9.2);                //输出 “float(9.2)”
    echo gettype(14.628);         //输出 “double”

    $value = "12";                //设定一个数字字符串

    //判断是否为 NULL
    if(is_null($value) == true){
        echo '$value 是一个 NULL 值';
    }

    //判断是否为有效的数字
    if(is_numeric($value) == true){
        echo '$value 是一个数字';
    }

    //判断是否为有效的数值
    if(is_nan(acos(14.01)) == true){
        echo 'acos(14.01) 的结果是一个无效的数值';
    }
?>
```

上述代码的执行结果如下。

```
float(9.2)
double
$value 是一个数字
acos(14.01) 的结果是一个无效的数值
```

10.2.2 强制类型转换

在许多情况下，为了满足各种程序接口的要求，需要进行必要的强制类型转换。在 PHP 中，强制类型转换可以使用以下三种方式。

- ❑ 语法结构转换
- ❑ 变量类型转换函数
- ❑ 使用 `sprintf()` 函数

1. 语法结构转换

在 PHP 的语法结构中，提供强制类型转换的机制。与其他语言类似，将目标类型的种类写在变量前面的圆括号中。允许的强制转换如下。

- ❑ `(int)`, `(integer)`: 转换成整型
- ❑ `(bool)`, `(boolean)`: 转换成布尔型
- ❑ `(double)`, `(float)`, `(real)`: 转换成浮点型
- ❑ `(string)`: 转换成字符串

- (array): 转换成数组
- (object): 转换成对象

代码示例如下。

```
<?php
$str      = '123.4abc';
$integer  = (int) $str;           // $integer = 123;
$double   = (float) $str;        // $double = 123.4
?>
```

2. 变量类型转换函数

还有三个更为具体的转换函数，即 `intval()`、`floatval()` 和 `strval()`。它们是分别用来转换整型、浮点型和字符串的。代码示例如下。

```
<?php
$string   = "123.4abc";
$integer  = intval($string);      // 转为整型, $integer = 123;
$double   = floatval($string);    // 转为浮点型, $double = 123.4;
$string   = strval($double);      // 转为字符串, $string = "123.4";
?>
```

另一个更为强大的类型转换函数是 `settype()`。它可以将一个变量转换为指定的数据类型。数据类型必须是以下类型之一。

- `boolean` (或为 `bool`)：布尔型
- `null`：空
- `integer` (或为 `int`)：整型
- `float`：浮点型
- `string`：字符串
- `array`：数组
- `object`：对象

如果类型转换成功则返回 `True`，失败则返回 `False`。下面是一段示例代码。

```
<?php
$value1   = "5star"; // 字符串
$value2   = true;    // 布尔值

settype($value1, "integer"); // $value1 现在是整型
settype($value2, "string");  // $value2 现在是字符串

echo $value1;           // $value1 现在是 5
echo $value2;           // $value2 现在是 "1"
?>
```

3. 使用 `sprintf()` 函数

如果要获得一个指定精度的浮点数，可以使用函数 `sprintf()`，其语法和 C 语言中的同名函数类似，可以返回一个格式化了了的字符串。另外，使用它对表达式的计算结果做一些清理工作是十分方便地。例如，下面是一段计算圆面积的程序。

```
<?php
if(!defined(M_PI)){
    define(M_PI, 3.1415826); // 定义圆周率
}

$radius = 19.1; // 半径
$area = M_PI * $radius * $radius; // 面积
```

```

echo $area;                                //结果为 1146.0844159561
echo sprintf("%.2f", $area);                //结果为 1146.08, 更易于阅读

$foo = (float) sprintf("%.2f", $area);      //如有必要, 可以进行类型转化
?>

```

10.2.3 字符串转化

PHP 所支持的各种类型都可以被转化为字符串。而在实际应用中, 字符串通常会被当作数字型值进行计算。下面将分别讲述。

1. 转化为字符串

NULL 将被转换成空字符串。

布尔值 True 将被转换为字符串“1”, 值 False 将被转换为“” (即空字符串)。这样就可以随意地在布尔值和字符串之间进行比较了。

对于整数或浮点数数值, 在转换成字符串时, 字符串即为表示这些数值的数字。当浮点数转换为字符串时, 还包含有指数部分, 如 3.21E4。对于一些无效的数值, 将转换为“-14.#IND”之类的字符串, 例如“(string) acos(14.01)”。

数组将被转换成字符串“Array”, 对象将被转换成字符串“Object”, 资源类型将以“Resourceid#1”的格式被转换成字符串 (其中 1 是 PHP 在运行时由资源指定的唯一标识)。

2. 字符串数值

字符串经常被当作数字型的值来计算, 如果这个字符串中包含有“.”、“E”或“e”, 那么它将被当作一个浮点型变量来处理, 否则将被当作一个整数。

在字符串中, 只有词首的部分被转换为数值。只要这个字符串以任何有效的数字数据开始, 那么表达式就以这个数字数据进行计算, 否则, 就以数字 0 替代。例如下面的代码。

```

<?php
$result = 0 + "10.5";                // $result = 10.5
$result = (float) "+5.6e4f3g2";       // $result = 5.6e4 = 56000
$result = floatval("-14.2abcd");      // $result = -14.2

$result = "xy1234" / 5;               // 运算结果是 0 / 5 = 0
$result = "4.3.2.1" * 5;              // 运算结果是 4.3 * 5 = 214.5
$result = 1 + "-14.3e3";              // 运算结果是 1+(-1300) = -1299
?>

```

注意：有效的数字数据遵循以下的原则, 跟随一个或多个数字后面 (可以包括小数点)、跟随在一个可选的指数后面。指数是由一个“e”或者“E”, 以及一个或多个数字组成。

10.2.4 数字转化

布尔值和 NULL 值都可以转换为数值型值。True 将转换为数字 1, 而 False 和 NULL 值都将转换为 0。这样就可以使一个数字与这些值进行任意的比较。

浮点数和整数类型间可以相互转化。整型转化为浮点型, 由于浮点型的精度范围远大于整型, 因此转化后的精度不会改变。

浮点型转化为整型, 将自动舍弃小数部分, 只保留整数部分。当一个浮点数超过整型数字的

有效范围时（整型的最大值约是 2.147×10^9 即 2^{31} ），其结果将是不确定的。这是因为溢出的部分已经丢失，无法给出一个确切的原始结果。例如下面的代码。

```
<?php
    $real_num = 3.1e9;           //定义一个超过整型范围的数

    //一个浮点数
    echo $real_num;              //输出 3100000000

    //浮点数转化为整型数
    echo (int) $real_num;        //输出一个不确定的值，可能的值为：-1194967296
?>
```

10.2.5 数组转化

将一个布尔值、整数、浮点数或字符串转化为数组型数据，将得到一个以上述类型数据为元素的数组（其下标为 0）。而将 NULL 转化为数组，会得到一个空数组。

将一个对象转换成一个数组，所得到的数组的元素为该对象的属性。其键名为成员变量名，元素的值为成员变量的值。例如，将前面“汽车”的对象转换为一个数组。

```
<?php
    $car = new Car("Mazda");      //创建一个 Car 对象，名字为 Mazda
    $car->setVelocity(10);         //设置初始速度为 10

    $arr_car = (array) $car;
?>
```

其输出结果是：

```
Car Object
(
    [brand] => Mazda
    [velocity] => 10
)
```

10.3 小结

本章主要介绍了 PHP 中的常用数据类型，以及这些类型之间的相互转化。PHP 支持的数据类型包括：布尔型、NULL 型、整型、浮点型、字符串型，以及数组、对象和资源型。

作为一种弱类型程序语言，PHP 不需要在声明变量时就确定变量的类型。一个变量的类型主要由它被实际使用时的上下文决定。数据类型间的转化，主要是为了兼容各种程序接口的需求，这也使得 PHP 的使用变得异常灵活。

第 11 章 常见 PHP 字符串处理

PHP 中提供了丰富的字符串处理函数，本章介绍了许多用于字符串处理的方法，包括字符串的整理、比较、查找替换及格式化的方式。最后，结合实际，介绍了在 HTML 页面和 URL 链接中使用字符串时需要注意的问题。

11.1 常见的 3 种字符串分析

PHP 把字符串看作一种基本的数据类型来处理。这与某些语言（如 C 语言）不同。因此，PHP 也提供了特定的函数对字符串进行分析。

11.1.1 访问字符串中的字符

在字符串分析时，最常用的运算是访问字符串中的字符。一般而言，如果知道了字符串的长度，以及目标字符所在位置，就可以对字符串进行分析。

1. 计算字符串的长度

PHP 中使用函数 `strlen()` 计算字符串的长度。这个函数比较简单，并且对二进制数据有效。与 C 语言不同，PHP 不会在字符串的末尾添加一个 “\0” 的结束标志。例如下面的代码。

```
<?php
$string = "Hello, my nice PHP world!";
echo strlen($string);           //输出 25
?>
```

2. 访问单个字符

在字符串中，可以使用字符串偏移量语法来定位单个字符。这种语法是在字符串变量后使用花括号指定所要字符的偏移量，偏移量从 0 开始。例如，“`$string{5}`”，是返回 `$string` 中第 5 个字符。下面是一个简单的范例。

```
<?php
$string = 'Hello';
//用循环的方式输出字符串中的每个字符
for ($i=0; $i < strlen($string); $i++)
{
    printf("第 %d 个字符是 “%s” \n", $i, $string{$i});
}
?>
```

程序的运行结果为。

```
第 0 个字符是 “H”
第 1 个字符是 “e”
第 2 个字符是 “l”
第 3 个字符是 “l”
第 4 个字符是 “o”
```

3. 检索字符的位置

使用 `strpos()` 和 `strrpos()` 函数可以计算子字符出现的位置，它们的函数原形如下。

```
int strpos (string $string, string $needle [, int $offset])
int strrpos (string $string, string $needle [, int $offset])
```

函数 `strpos()` 用于查找子字符串 `$needle` 在目标字符串 `$string` 中第一次出现的位置，而函数 `strrpos()` 用于查找某个字符（注意是单个字符而不是字符串）在字符串中最后一次出现的位置。可选的 `$offset` 参数是目标字符串的偏移量，它确定了子字符串的检索是从 `$offset` 位置开始的。例如下面的代码。

```
<?php
$string = "sexy sexy";
$first_positon = strpos($string, "x");      //从首部计算，返回 2
$last_positon = strrpos($string, "x");      //从尾部计算，返回 7
$positon = strpos($string, "s");           //从首部计算，返回 0
$positon = strrpos($string, "y", 2);        //从尾部计算，返回 8
?>
```

需要注意的是，如果没有查询到子字符出现的位置，则将返回 `False`。因此，判断一个字符串中是否包含另一个子串的正确方法是使用恒等号“`==`”。例如下面的代码。

```
<?php
$pos = strpos($string, "s");
if($pos === false){      //使用“===”判断
    //没有找到字符
}
?>
```

11.1.2 处理子字符串

子字符串(Substrings)是截取整个字符串的全部或者一部分。PHP 中使用 `substr()`、`substr_count()` 和 `substr_replace()` 等函数来处理子字符串。

1. 截取子字符

函数 `substr()` 用来截取子字符串，其原形如下。

```
string substr(string $string, int $start [, int $length])
```

该函数从字符串 `$string` 的 `$start` 位置处截取字符。可选的 `$length` 参数控制返回的子字符串长度。如果 `$length` 没有给定，则 PHP 将截取从 `$start` 位置开始到字符串 `$string` 结尾的全部字符。例如下面的代码。

```
<?php
$string = "Just do it! Practice makes perfect.";

$practice = substr($string, 12, 9);          //返回“Practice”
$practice = substr($string, -23, 9);         //返回“Practice”

$last = substr($string, 26, 8);              //返回“perfect.”

echo substr($string, -strlen($string), 11); //输出“Just do it!”
echo substr($string, -1);                   //输出“.”
?>
```

2. 查找字符串出现的次数

函数 `substr_count()` 用来查找子字符串出现的次数，其原形如下。

```
int substr_count (string $string, string $substring [, int $start [, int $length]])
```

该函数从字符串\$string 中查找子字符串\$substring 出现的次数，可选的\$start 参数指定了在\$string 字符串中查询的起始位置，可选的\$length 参数指定了允许查询的最大长度，该函数是大小写敏感的。例如下面的代码片断。

```
<?php

$genesis = <<< BIBLE_GENESIS

In the beginning God created the heaven and the earth.
And the earth was without form, and void; and darkness
was upon the face of the deep. And the Spirit of God
moved upon the face of the waters.
And God said, Let there be light: and there was light.

BIBLE_GENESIS;

$god_number = substr_count($genesis, "God");
$and_number = substr_count($genesis, "and");

echo $god_number; //返回 3
echo $and_number; //返回 4

?>
```

3. 替换子字符串

函数 substr_replace()允许使用不同的方式替换子字符串。该函数的原形如下。

```
string substr_replace ($string, $new, $start [, $length ])
```

该函数用字符串\$new 替换\$string 的一部分。这一部分由参数\$start 和\$length 决定。如果没有给出\$length 参数，则 substr_replace()会删除从\$start 到字符串末尾的字符。

应用 substr_replace()是十分灵活和方便的，例如下面的代码。

```
<?php
$string = "I was in heaven when I heard the news.";

//将 “in heaven” 替换为 “so happy”
$result = substr_replace($string, "so happy", 6, 9);
//返回的结果为 “I was so happy when I heard the news.”

//如果$start 为负值，则指定从字符串末尾开始到字符串开头替换的字符数；
//如果$length 为 0，则实现无删除的插入
$result = substr_replace($result, "good ", -5, 0);
//返回的结果为 “I was so happy when I heard the good news.”

//设定$new 为 “”，来实现无插入的删除
$result = substr_replace($result, "", 14);
//返回的结果为 “I was so happy”

//如果$length 为负值，则指定从字符串末尾开始删除的字符个数
$result = substr_replace($result, "", -8, -5);
//返回的结果为 “I was happy”

//在字符串的开头插入内容
$result = substr_replace($result, "It was my birthday and ", 0, 0);
//返回的结果为 “It was my birthday and I was happy”

?>
```


11.1.3 分割字符串

将一个字符串分割为多个部分，可以有多种方式。可以按照特定的字符（如空格、“|”或“，”等），也可以按照指定的数目，对字符串进行平均分割。

1. 按照特定字符进行分割

函数 `explode()` 可以将字符串按照特定的字符分割，并把结果保存在数组当中。`explode()` 的函数原形如下。

```
array explode ( string $separator, string $string [, int $limit])
```

此函数返回由字符串组成的数组，每个元素都是 `$string` 的一个子串，它们被字符串 `$separator` 作为边界点分割出来。如果设置了 `$limit` 参数，则返回的数组包含最多 `$limit` 个元素，而最后那个元素将包含 `$string` 的剩余部分。

下面是一个用逗号进行字符串分割的例子。

```
<?php
$pizza = "一片,两片,三四片,五片,六片,七八片";
$pieces = explode(",", $pizza);
echo $pieces[0];           //输出 “一片”
echo $pieces[1];           //输出 “两片”
echo $pieces[4];           //输出 “六片”
echo $pieces[5];           //输出 “七八片”

$pieces = explode(",", $pizza, 5);
echo $pieces[4];           //输出 “六片,七八片”
?>
```

`explode()` 另一个典型的应用是与 `list()` 配合，直接将数组转化为字符串。例如下面的代码。

```
<?php
$birthday = "1980-7-14";
list($year, $month, $day) = explode("-", $birthday);

echo $year;           // 1980
echo $month;          // 7
echo $day;            // 14
?>
```

`split()` 函数与 `explode()` 函数类似，但其功能更加强大。`split()` 函数可以使用一个模式表达式对字符串进行分割。关于模式表达式的内容可以参考本书的第 3 章。这里仅举一个简单的例子，在下面的代码中，分割符 “-|:” 就是一个模式表达式，它代表以 “-”、空格或者 “:” 进行分割。

```
<?php
$birthday = "1982-6-7 23:40:56";
list($year, $month, $day, $hour, $minute, $second) = split("-|:", $birthday);

echo $hour;           // 23
echo $minute;         // 40
echo $second;         // 56
?>
```

2. 按照指定数目进行分割

函数 `str_split()` 可以将字符串按照指定数目的字符平均分割，并将结果保存在数组当中。`str_split()` 的函数原形如下。

```
array str_split (string $string [, int $split_length])
```

此函数返回由字符串组成的数组。字符串 `$string` 将在每隔 `$split_length` 个字符的位置进行一次分割。如果 `$split_length` 参数没有设置或者为 0 值，将依次输出 `$string` 中的每一个字符。例如

下面的代码。

```
<?php
    $string = "Happy Birthday";

    $array1 = str_split($string);           //依次分割每个字符
    $array2 = str_split($string, 3);       //每隔 3 个字符，分割一次

    print_r($array1);
    print_r($array2);
?>
```

返回结果为。

```
Array
(
    [0] => H
    [1] => a
    [2] => p
    [3] => p
    [4] => y
    [5] =>
    [6] => B
    [7] => i
    [8] => r
    [9] => t
    [10] => h
    [11] => d
    [12] => a
    [13] => y
)
Array
(
    [0] => Hap
    [1] => py
    [2] => Bir
    [3] => thd
    [4] => ay
)
```

11.2 4 种字符串的运算

对字符串进行整理，包括一系列运算，如删除字符串两边的空白、填充字符串到固定长度、转换单词的大小等。下面将逐一介绍。

11.2.1 删除字符串的空白

可以使用函数 `trim()`、`ltrim()` 和 `rtrim()` 删除字符串的开头或结尾的字符，它们的函数原形如下所示。

```
string trim (string $string [, string $charlist])
string ltrim (string $string [, string $charlist])
string rtrim (string $string [, string $charlist])
```

参数 `$charlist` 是可选的，它指定所要删除的字符，默认是删除空白字符。`trim()` 函数将删除字符串 `$string` 开头和结尾的空白。`ltrim()` (l 就是 left) 函数将删除字符串开头 (左侧) 的空白，`rtrim()` (r 就是 right) 函数将删除字符串结尾 (右侧) 的空白。所谓“空白字符表”，如表 11.1 所示。

表 11.1 空白字符表

字符名称	转义字符	ASCII码值（十六进制）
空格		32 (0x20)
制表符	\t	9 (0x09)
回车符	\r	13 (0x0D)
换行符	\n	10 (0x0A)
垂直制表符	\v	11 (0x0B)
NULL	\0	0 (0x00)

例如：

```
<?php
$string = "   Hello PHP World!  \n\t ";

$stringed = trim($string);           //删除开头和结尾的空白
var_dump($stringed);

$stringed = ltrim($string);          //删除开头的空白
var_dump($stringed);

$stringed = rtrim($string);          //删除结尾的空白
var_dump($stringed);
?>
```

程序的返回结果如下。

```
string(16) "Hello PHP World!"
string(20) "Hello PHP World!
"
string(19) "   Hello PHP World!"
```

使用 `trim()` 等函数只能删除单字节的空白字符，即表 11.1 中说明的那些。如果要删除诸如“全角空格”等字符时，这些函数是不起作用的。字符串中的空白字符，也并不仅仅局限于表 11.1 中的几个。如果必要，应该在删除列表 `$charlist` 中指明那些多余的字符。

例如，下面的代码要求删除字符串两侧除了制表符（\t）以外的空白，以及多余的“.”。

```
<?php
$string = "\n\tHello PHP World! ... \n";

$stringed = trim($string, ". \r\n\0\x0B");
var_dump($stringed);
?>
```

程序的输出结果为。

```
string(17) "   Hello PHP World!"
```

11.2.2 转换字符串大小写

关于字符串大小写的转换函数，主要有以下 4 个。

- ❑ `strtoupper()` 将给定字符串全部转换为大写字母。
- ❑ `strtolower()` 将给定字符串全部转换为小写字母。
- ❑ `ucfirst()` 将给定字符串的首字母转换为大写。
- ❑ `ucwords()` 将给定字符串中全部单词的首字母转换为大写。

下面是这些函数使用的一些程序片断。

```
<?php
$str = "Where is the film showing? ";
```

```

$lower = strtolower($str);           //全部转换为小写
echo $lower . "\n";

$upper = strtoupper($str);           //全部转换为大写
echo $upper . "\n";

$string = 'hello PHP world!';
$ucfirst = ucfirst($string);         //将整句的首字母转为大写
echo $ucfirst . "\n";

$string = "i try do some programming/repairing.";
$ucwords = ucwords($string);         //将单词的首字母转为大写
echo $ucwords . "\n";
?>

```

代码的运行结果如下。

```

where is the film showing?
WHERE IS THE FILM SHOWING?
Hello PHP world!
I Try Do Some Programming/repairing.

```

这些函数只是按照它们说明中描述的方式工作。因此，要想确保一个字符串的首字母是大写字母，而其余的是小写字母，则应使用复合的方法。例如下面的代码。

```

<?php
$string = ' hello PHP world!';
echo ucfirst($string);           //输出“Hello PHP world!”
echo ucfirst(strtolower($string)); //输出“Hello php world!”
?>

```

11.2.3 填补字符串

可以使用函数 `str_pad()` 进行字符串的填补，其原形如下。

```
string str_pad (string $string, int $length [, string $pad_str [, int $pad_type]])
```

`str_pad()` 有 4 个参数。`$string` 参数是要处理的字符串。`$length` 参数指定了处理后字符串的长度。`$pad_str` 参数给出了填补所需的字符串，默认使用空格进行填补。`$pad_type` 参数控制填补的方向，有以下三个可选值。

- ❑ `STR_PAD_LEFT`，在字符串左侧（开头）进行填补，是默认值。
- ❑ `STR_PAD_RIGHT`，在字符串右侧（结尾）进行填补。
- ❑ `STR_PAD_BOTH`，在字符串两端进行填补。

例如，下面的程序是用星号打印一个“金字塔”。其原理是在每一行星号的两侧填补空格，使它们达到统一的长度，从而形成美观的“金字塔”形状。

```

<?php
$start = "*";
for($i=0; $i<5; $i++)
{
    $line = str_repeat($start, $i*2+1);           //重复打印 ($i*2+1) 个星号

    echo str_pad($line, 11, " ", STR_PAD_BOTH);
    echo "\n";
}
?>

```

上述程序的运行结果如下。

```

*
```

```

    * * *
  * * * * *
* * * * * * *
* * * * * * * *

```

11.2.4 反转字符串

至少有两种方式可以实现字符串的反转，一种是使用 `strrev()` 函数，它的反序是按照字节进行的，因此无法处理双字节字符。例如下面的代码。

```

<?php
    echo strrev("Hello world!");    //输出 “!dlrow olleH”
?>

```

另一种方式是读取字符串中的每个字节，然后倒序组成一个新的字符串。这种方式的好处是可以控制反转过程的细节，因此可以处理多字节字符。下面是一段简单的双字节字符反转程序。

```

<?php

$string = "北京欢迎你";
$strrev = "";

$length = strlen($string);

for($i=0; $i<=$length-2; $i+=2)
{
    $first = $length - $i - 2;
    $second = $length - $i - 1;

    $strrev .= $string{$first} . $string{$second};
}

echo $strrev;    //输出 “你欢迎京北”
?>

```

11.3 2 种字符串的格式化

在实际开发中，经常需要将数字转换为以“,”进行分割的数字，这时就需要使用 `number_format()` 函数进行转换。PHP 中提供了 `printf()` 和 `sprintf()` 函数，可以对数字或字符串等进行其他的格式化运算。这两个函数源自 C 语言标准函数库中的同名函数。

11.3.1 格式化数字

按照惯例，在表示金额或其他数字时，通常将数字的整数部分使用千分进位符“,”进行分隔。如“1234.56”应该表示为“1,234.56”。为实现这一转换，可以使用 `number_format()` 函数。它有两种使用形式，其原形如下。

```

string number_format (float $number [, int $decimals])
string number_format (float $number, int $decimals, string $dec_point, string
$thousands_sep)

```

`number_format()` 函数的第一种形式就能够实现这种国际通行的表示方式。其中，参数 `$decimals` 指定数字的小数部分的位数。

例如，下面是 `number_format()` 函数的第一种应用形式的程序示例。

```
<?php
    $number = 12345.6789;
    $internet_format = number_format($number);           //返回 “12,345”
    $internet_format = number_format($number, 2);         //返回 “12,345.67”
?>
```

在 `number_format()` 函数的第二种形式中，还可以接受另外两个参数。参数 `$dec_point` 表示整数部分与小数部分的分界符号，默认为小数点 “.”；参数 `$thousands_sep` 表示千分进位符，默认为逗号 “,”。这种用法，可以用于其他地区的数字表示方式。

例如，下面的代码可以用于表示欧洲某些地区的数字格式，他们习惯使用空格作为千分进位符、用 “,” 作为小数点，如 “1 234,50”。

```
<?php
    $number = 1234.5;
    $european_format = number_format($number, 2, ',', ' '); //返回 “1 234,50”
?>
```

11.3.2 格式化字符串

函数 `printf()` 和 `sprintf()` 源自 C 语言标准函数库中的同名函数，二者功能相同，都是将值传递到一个字符串模板（即定义好的格式字符串）上进行格式化。不同的是 `printf()` 函数会打印字符串格式化后的结果，而 `sprintf()` 函数只是将其返回。

函数 `printf()` 和 `sprintf()` 的第一个参数是格式字符串，其他的参数是要替换进来的值。在格式字符串中，由 “%” 字符指定一个替换。

1. 格式修饰符

在模板中，每一个替换标记都由一个百分号 “%” 组成，其后面可能跟着一个格式修饰符（见表 11.1），并以类型说明符结尾。修饰符必须按下面的次序出现。

- ❑ 填充说明符。表示该字符用于填充结果，使结果为适当长度的字符串。规定其为 0、一个空格或任何以一个单引号为前缀的字符。默认情况下用空格填充。
- ❑ 加号 “+” 或减号 “-”。对于字符（串），表示左对齐或右对齐；对于数字，表示正数或负数。默认使用 “+” 表示。
- ❑ 整数。如果字符串的长度少于该值，则使用前面的填充说明符填充到这个长度。
- ❑ 小数点加一个整数。对浮点数字，精确说明应该保留的小数位。除此之外，忽略这个说明符。

2. 类型说明符

类型说明符会告诉 `printf()` 和 `sprintf()` 函数，什么样的数据类型将被替换。表 11.2 列出了 PHP 中支持的类型说明符及其含义。

表 11.2 类型说明符及其含义

类型说明符	含义
%	打印百分号 “%”，不转换
b	替换为一个二进制的数字
c	替换为一个字符
d	替换为一个整型的数字
e	替换为一个浮点型的数字（科学计数表示）

续表

类型说明符	含义
f	替换为一个浮点型的数字
u	替换为一个长整型的数字
o	替换为八进制数字
s	替换为字符串
x	替换为小写十六进制数字
X	替换为大写十六进制数字

下面是一些关于 `sprintf()` 函数的应用范例。

```
<?php
//数字的表示
$money1 = 9123.453;
$money2 = 678.9;
$sum_money = sprintf("¥%01.2f", $money1 + $money2); //返回“¥98015.35”

//日期的表示
$date = sprintf("%04d年%02d月%02d日", 2007, 7, 14); //返回“2007年07月14”

//一个百分比
echo sprintf("%.2f%", 315.1); //输出 315.10%

//填充整数，使长度达到 3 位
$format = "%s %03d";
echo sprintf($format, "Bond. James Bond", "7");//输出“Bond. James Bond 007”

//十进制和十六进制
$num = 1234;
echo sprintf("The hex value of %d is %X", $num, $num);
//输出“The hex value of 1234 is 4D2”

//科学计数法
$number = 362543200;
echo sprintf("%.3e", $number); //输出 3.63e+8
?>
```

下面是关于 `printf()` 函数的应用范例，它演示了如何设置字符串的显示格式。使用这里提到的方法，可以对字符串进行最大程度的修饰。

```
<?php
$string = "Windows";
$test = "Many CEO groups";

printf("|%s|\n", $string); //标准的字符输出
printf("|%10s|\n", $string); //右对齐，使用空格填补
printf("|%-10s|\n", $string); //左对齐，使用空格填补
printf("|%010s|\n", $string); //右对齐，用数字 0 填补空缺
printf("|%'#-10s|\n", $string); //左对齐，用自定义字符“#”填补空缺
printf("|%10.10s|\n", $test); //右对齐，并将字符串截短到 10 个字节
?>
```

程序的执行结果如下。

```
|Windows|
| Windows|
|Windows |
|000Windows|
```

```
|Windows###|
|Many CEO g|
```

11.4 字符串的查找和替换

PHP 提供了强大的字符串处理功能，其中包括一组函数，用以完成对字符串的查找和替换运算。下面将详细介绍。

11.4.1 查找字符串

前面提到的 `strpos()` 和 `substr()` 函数都可以实现字符串的查找。此外，PHP 还提供了一系列功能更强大的函数，主要有 `strstr()`、`strspn()` 和 `strpbrk()` 等。

1. 查找子字符串

函数 `strstr()` 用于查找一个子字符串。函数 `stristr()` 与其功能相同，只是对字符的大小写并不敏感。它们的函数原形如下所示。

```
string strstr (string $string, string $needle)
string stristr (string $string, string $needle)
```

这两个函数都是从字符串 `$string` 中寻找 `$needle` 第一次出现的位置，并返回这个位置以后的子字符串，如果没有找到就返回 `False`。例如下面的代码。

```
<?php
    $string = "The Golden Global View";           //字符串
    $mystr = "obal";                             //要查找的字符串

    $found = strstr($string, $mystr);
    if($found){
        echo $found;
    }else{
        echo "没有发现 $found";
    }

    //输出结果为“obal View”
?>
```

2. 搜寻出现的字符

函数 `strspn()` 用于在一个字符串中搜寻另一个字符串中所出现的字符。其函数原型如下所示。

```
int strspn (string $str1, string $str2)
```

该函数是在字符串 `$str1` 中搜寻 `$str2` 中所出现的字符，并返回出现在 `$str1` 中的字符的总数。函数 `strspn()` 的搜索是从 `$str1` 的第一个字符开始的。例如下面的代码。

```
<?php
    $string1 = "hello world!";
    $string2 = "abcdefghij";

    $found = strspn($string1, $string2);          //返回 2
?>
```

上述代码返回值为 2，因为从 `$string1` 字符串的首字符开始，只有“he”两个字符同时在 `$string1` 和 `$string2` 中出现。再如：

```
<?php
    echo strspn("163.com", "9876543210");        //返回 3
?>
```


3. 字符匹配

函数 `strpbrk()` 用于在一个字符串中查找与另一个字符串的匹配情况。其函数原型如下所示。

```
string strpbrk (string $string, string $char_list)
```

该函数是在字符串 `$string` 中搜寻与 `$char_list` 中任何一个字符相匹配的第一个字符的位置，并返回这个位置之后的字符串。如果没有找到可匹配的项目，就返回 `False`。例如下面的代码。

```
<?php
    $string = 'This is an Example string.';

    echo strpbrk($string, 'mis');
?>
```

上述代码的执行结果是：

```
is is an Example string.
```

可以看出字符串 “mis” 中的 “i” 第一次在单词 “This” 中出现，所以输出的结果为 “is is an Example string.”。

11.4.2 替换字符串

PHP 提供了强大的字符串替换功能，可以通过使用 `str_replace()` 和 `strtr()` 两个函数实现。它们都可以将一个字符串替换为另一字符串。

1. 字符串替换

函数 `str_replace()` 用于字符替换，主要有三种用法，其函数原型如下所示。

```
string str_replace(string $search, string $replace, string $string)
string str_replace(array $search, string $replace, string $string)
string str_replace(array $search, array $replace, string $string)
```

函数 `str_replace()` 的功能是将字符串 `$string` 中的 `$search` 部分用 `$replace` 替换。其中，参数 `$search` 和 `$replace` 可以是字符串，也可以是字符串数组。

第一种用法是最基本的用法，其中，参数 `$search` 和 `$replace` 都是字符串，函数将使用 `$replace` 替换所有的 `$search`。例如，下面的代码将 HTML 中的特殊字符替换为实体字符。

```
<?php
    $template = <<< TPL
<a href="links.php">链接 地址</a>
软件下载 100
TPL;

    $template = str_replace("<", "&lt;", $template);           //替换 "<"
    $template = str_replace(">", "&gt;", $template);           //替换 ">"
    $template = str_replace(" ", "&nbsp;", $template);         //替换空格
    $template = str_replace("\n", "\n<br>", $template);       //替换空格

    echo $template; //输出替换后的结果
?>
```

上述程序的输出结果如下所示，尽管这样的结果往往令人难以阅读，但在很多情况下，这种用法是十分安全的。

```
&lt;a&nbsp;href="links.php"&gt;链接&nbsp;地址&lt;/a&gt;
<br>软件下载&nbsp;100
```

在第二种用法中，参数 `$search` 是一个字符串的数组。函数将使用同一个 `$replace` 字符串替换数组 `$search` 中的每一个元素。例如，下面的代码将字符串中指定的数字替换为 “?”。

```
<?php
    $evens = array(2, 4, 6, 8, 0);    //数字数组
```

```
$string = "In 1998, my 43 years-old father earn 3,560,289 yuan.";
echo str_replace($evens, "*", $string);
//输出结果 "In 199*, my *3 years-old father earn 3,5**,**9 yuan."
?>
```

在最后一种用法中，`$search` 和 `$replace` 均为字符串的数组。通常，其中的元素彼此对应。函数将用 `$replace` 中的元素替换 `$search` 中的元素。例如，下面的代码演示了颜色数组的替换。

```
<?php
//颜色代码数组
$codes = array(
    "%back%",
    "%red%",
    "%green%",
    "%yellow%",
);

//颜色值数组
$colors = array(
    "#000000",
    "#FF0000",
    "#008000",
    "#FFFF00",
);

//字符串
$string = <<< HTML
<body backgroundd="%yellow%">
<font color="%back%">黑色</font>
<font color="%green%">绿色</font>
</body>
HTML;

$string = str_replace($codes, $colors, $string);
echo $string;
?>
```

程序的输出结果如下：

```
<body backgroundd="#FFFF00">
<font color="#000000">黑色</font>
<font color="#008000">绿色</font>
</body>
```

2. 字符串翻译

函数 `strtr()` 是另一个重要的字符串替换函数，也叫字符串翻译函数。`strtr()` 函数有两种调用形式，其语法结构如下所示。

```
string strtr(string $string, string $from, string $to)
string strtr(string $string, array $replace_pairs)
```

第一种是 `strtr()` 的基本调用形式，函数将返回一个 `$string` 字符串的备份。其中，在 `$from` 中的每一个字符都将被 `$to` 中对应的字符替换。例如下面的代码。

```
<?php
$string = 'Welcome to Blog.taodoor.com!';
$string = strtr($string, 'Blog', 'Pair');
echo $string;
?>
```

程序的输出结果为：

```
Weacime ti Pair.taidiir.cim!
```

从上述程序可以看出，不仅是“Blog”被替换为“Pair”，字符串\$string中的“B”、“l”、“o”和“g”也分别被字符“P”、“a”、“i”和“r”替换。

注意：strtr()函数的功能不是将\$from字符串替换为\$to，而是将它们中的对应字符进行替换。如果设置的\$from和\$to字符串长度不一样，那么将以二者中较短的那个字符串长度为准，并把较长的那个字符串长度截短。

第二种是strtr()的扩展调用形式，函数将返回一个\$string字符串的备份。其中，\$replace_pairs数组的每个元素的键名和值都是一个替换对，并彼此替换。例如下面的代码。

```
<?php
$string = 'Welcome to Blog.taodoor.com!';
$change = array( 'Blog'=>'www', 'com'=>'net');    //字符串翻译数组
$string = strtr($string, $change);                //进行替换
echo $string;
?>
```

程序的输出结果为：

```
Welnete to www.taodoor.net!
```

11.5 3 种常见的字符串的比较方法

有多种方法可以比较两个字符串的大小。最常见的方法是直接使用条件运算符加以比较，还可以按ASCII码顺序比较、按“自然排序”法比较或按相似性比较等。

11.5.1 按ASCII码顺序比较

使用字符串比较函数strcmp()，可以对字符串按照ASCII码的顺序进行比较。其函数原型如下所示。

```
int strcmp (string $str1, string $str2)
```

该函数返回一个整数。放入strcmp()的两个字符串将按照字节的ASCII码值进行比较。比较结果有三种情况。

- \$str1>\$str2，返回一个负数。
- \$str1<\$str2，返回一个正数。
- \$str1=\$str2，返回0。

例如下面的代码。

```
<?php
$name_1 = "Thomas";
$name_2 = "Tom";

if(strcmp($name_1, $name_2)>0){
    echo "{$name_1} 在前面";
}else{
    echo "{$name_2} 在前面";
}

//输出结果为“Tom 在前面”
?>
```

函数strcasecmp()是函数strcmp()的一个变种。函数strcasecmp()在比较时忽略字符串中的大小

写。也就是说，同一个字母的大小写形式认为是相等的。例如：

```
<?php
    $str1 = "php LanGuaGe";
    $str2 = "PHP Language";
    $test = strcasecmp($str1, $str2);                                //返回 0
?>
```

实际上可以这样认为，`strcasecmp()`函数是先将两个字符串全部转化为小写（或者大写），然后再使用 `strcmp()`函数进行比较，其过程如下。

```
<?
    //下面的比较是等效的
    $test =strcmp(strtolower($str1),strtolower($str2));                //返回 0
?>
```

在字符串比较时，可以只比较字符串的前几个字符。`strncmp()`和 `strncasecmp()`函数可以实现这个功能。它们具有一个附加的参数，这个参数可用来指定用于比较的初始字符个数。

```
int strncmp (string $str1, string $str2, int $offset)
int strncasecmp (string $str1, string $str2, int $offset)
```

例如：

```
<?php
    $n = strncmp("sample","sam",4);
    echo $n;                                //返回 1

    $n = strncasecmp("sample","Sam", 3);
    echo $n;                                //返回 0
?>
```

11.5.2 按“自然排序”法比较

在按照 ASCII 码顺序比较时，数字“2”是排在“12”后面的，这不太符合人类的思维习惯。因为按照“自然排序”法，“2”在“12”的前面。

“自然排序”法更加简单、灵活，它直接模拟了人类的思维习惯。如表 11.3 所示，人们会自然地吧文件“Picture9.png”排在“Picture10.png”的前面。但如果是按照 ASCII 码顺序比较的方法，则结果是完全不同的。

表 11.3 不同方法的字符串排序

按“自然排序”法比较	按ASCII码顺序比较
Picture1.png	Picture1.png
Picture15.png	Picture10.png
Picture9.png	Picture115.png
Picture10.png	Picture15.png
Picture115.png	Picture9.png

PHP 也提供了这种按照“自然排序”法比较字符串的函数 `strnatcmp()`，并且提供了 `strnatcasecmp()`函数作为不区分大小比较的版本。“自然排序”法会比较字符串中的数字部分，将字符串按照数字大小顺序进行比较。

例如，下面是一段从数组中取出“最大”文件名的代码：

```
<?php
    $files = array(
        "Picture1.png",
        "Picture15.png",
        "Picture10.png",
    );
```

```

        "Picture115.png",
        "Picture9.png",
    );

    $common_max = $files[0];
    $nature_max = $files[0];
    for($i=1; $i< count($files); $i++){
        //ASCII 码排序法比较
        if(strcmp($files[$i], $common_max)>0){
            $common_max = $files[$i];
        }
        //自然排序法比较
        if(strnatcmp($files[$i], $nature_max)>0){
            $nature_max = $files[$i];
        }
    }

    echo "ASCII 码排序法: " . $common_max;
    echo "“自然排序”法: " . $nature_max;
?>

```

下面是程序的运行结果，由此也可以看出两种比较方法的不同。

```

ASCII 码排序法: Picture9.png
“自然排序”法: Picture115.png

```

11.5.3 按相似性比较

在模糊查询或者一些较为“宽松”的查询中，计算字符串（主要是英文单词）的相似读音是非常有用的。PHP 提供了一些函数来测试两个字符串是否近似相等，如 `soundex()`、`metaphone()` 和 `similar_text()` 等。

这些函数的语法结构如下所示。

```

string soundex (string $str)
string metaphone (string $str)
int similar_text (string $first, string $second [, float &$percent])

```

1. 取得单词的发音

函数 `soundex()` 和 `metaphone()` 都是生成一个字符串，来表示单词在英语中的发音，我们可以用返回的字符串来比较两个单词的近似性。相对而言，`soundex()` 函数的比较更为苛刻些，也就是说，用 `metaphone()` 函数比较更容易得出相似的结果。例如下面的代码。

```

<?php
    $known = "Frebe";
    $query = "Phrebe";

    //soundex 的比较
    if (soundex($known) == soundex($query)) {
        echo "Soundex: $known 类似于 $query<br>";
    } else {
        echo "Soundex: $known 不同于 $query<br>";
    }

    //metaphone 的比较
    if (metaphone($known) == metaphone($query)) {
        echo "Metaphone: $known 类似于 $query<br>";
    } else {

```

```
        echo "Metaphone: $known 不同于 $query<br>";
    }
?>
```

下面是程序的运行结果。

```
Soundex: Coorebe 不同于 Kurebe
Metaphone: Coorebe 相似于 Kurebe
```

2. 查看单词的相似度

函数 `similar_text()` 用于查看两个单词的相似程度。该函数作用于两个字符串，返回它们中匹配的字符个数。也可以给出一个 `$percent` 变量，用于返回一个相似度的值。例如下面的代码。

```
<?php
    $string1 = "guest";
    $string2 = "ghost";

    $same_num = similar_text($string1, $string2, $similar);
    printf("%s 和 %s 中有 %d 个字符相同\n", $string1, $string2, $same_num);
    printf("%s 和 %s 的相似度: %d%%\n", $string1, $string2, $similar);
?>
```

下面是程序执行结果。

```
guest 和 ghost 中有 3 个字符相同
guest 和 ghost 的相似度: 60%
```

11.6 处理 HTML 和 URL

HTML 和 URL 是 Web 开发中的重要部分，因此，对它们的处理也格外重要。对 HTML 页面的处理主要包括：HTML 实体的处理、HTML 标签的清理等。对 URL 的处理主要包括：URL 字符串的解析、URL 编码处理和查询字符串的构造等。

11.6.1 HTML 标签的清理

在某些场合下，需要强制删除 HTML 中的标签，因为这是多余的。使用 `strip_tags()` 函数可以删除字符串中的 HTML 标签，其函数原型如下所示。

```
string strip_tags(string $html_string [, string $allowable_tags])
```

该函数会删除字符串 `$html_string` 中的 HTML 标签。参数 `$allowable_tags` 是一个可选的 HTML 标签列表，放入该列表中的标签将予以保留，不会被删除。

例如下面的代码，它允许显示一些特定的 HTML 效果（如粗体、下画线等），而屏蔽一些复杂的效果（如 `<table>` 表格）。

```
<?php
    $text = <<< HTML_TEXT
<table border="1">
    <tr><td>
        <p>段落标记: <b>粗体</b>字</p>
    </td><td>
        <!-- 注释文字 -->
        其他的文字, <i>斜体</i>字
    </td></tr>
</table>
HTML_TEXT;
```

```
echo strip_tags($text);           //删除所有标签
echo "\n-----\n";
echo strip_tags($text, '<i><b>');    //允许标签<i>, <b>
?>
```

下面是程序的输出结果。

```
段落标记: 粗体字
其他的文字, 斜体字
-----
段落标记: <b>粗体</b>字
其他的文字, <i>斜体</i>字
```

11.6.2 HTML 实体的处理

HTML 实体 (Entity) 是为了在浏览器中显示一些特殊字符, 而使用的转义字符。例如, 要想显示 “<”, 就必须使用代码 “<”, 因此, “<” 就是 “<” 的实体代码。再如, “&” 是 “&” 的实体代码。

在 PHP 中, 可以使用函数 htmlspecialchars () 和 htmlentities () 进行实体字符的转换, 它们的函数原型如下所示。

```
string htmlspecialchars (string $string [, int $quote_style [, string $charset]])
string htmlentities (string $string [, int $quote_style [, string $charset]])
```

这两个函数都会返回一个经过整理的字符串。函数 htmlentities () 可以将所有非 ASCII 码字符转换为对应的实体代码, 而函数 htmlspecialchars () 只转换下面几个特殊字符。

- ❑ “&” 转换为 “&”
- ❑ “” 转换为 “"”
- ❑ “'” 转换为 “'”
- ❑ “<” 转换为 “<”
- ❑ “>” 转换为 “>”

在参数列表中, 参数 \$quote_style 决定了字符串 \$string 中引号的转换方式, 它具有以下三种处理方式。

- ❑ ENT_COMPAT: 将只转换双引号, 而保留单引号。这是默认值。
- ❑ ENT_QUOTES: 将同时转换这两种引号。
- ❑ ENT_NOQUOTES: 将不对引号进行转换。

参数 \$charset 用来指定所处理的字符串 \$string 的字符集, 默认的字符集是 “ISO8859-1”。PHP 也支持更多 HTML 字符集, 如表 11.4 所示。

表 11.4 函数 htmlspecialchars () 和 htmlentities () 使用的合法字符集

字符集	别名	说明
ISO-8859-1	ISO8859-1	西欧, 拉丁文 Latin-1 字符
ISO-8859-15	ISO8859-15	西欧, 拉丁文 Latin-9, 以及未包含在拉丁文 Latin-1 (ISO-8859-1) 中的欧文字号、法语及芬兰与字母
UTF-8		与 ASCII 码符兼容的 8 位多字节 Unicode
cp866	ibm866, 866	针对 DOS 的 Cyrillic 字符
cp1251	Windows-1251, win-1251, 1251	针对 Windows 的 Cyrillic 字符
cp1252	Windows-1252, 1252	针对 Windows 的西欧字符

续表

字符集	别名	说明
KOI8-R	koi8-ru, koi8r	俄罗斯语
GB2312	936	简体汉字，主要用于中国大陆
BIG5	950	繁体汉字，主要用于中国台湾地区
BIG5-HKSCS		繁体汉字，主要用于中国香港特别行政区
Shift_JIS	SJIS, 932	日文字符
EUC-JP	EUCJP	日文字符

函数 `htmlspecialchars()` 的一个典型的应用就是，在 Form 表单中重写输入框的值，以提高安全性。例如下面的表单，在 `<Input>` 和 `<Textarea>` 域中使用 `htmlspecialchars()` 函数重写了 POST 传递的值。

```
<form action="test.php" method="post">
    标题: <input type="text" name="title"
        value="<?php echo htmlspecialchars($_POST["title"]);?>"
    <br>正文: <textarea name="content" cols=25 rows=10>
        <?php echo htmlspecialchars($_POST["content"]);?>
    </textarea>
    <br><input type="submit" name="submit" value="提交">
</form>
```

下面是 `htmlentities()` 的一些例子。

```
<?php
//特殊字符的转换
$str = "The 'quote' is <b>bold</b>";
echo htmlentities($str);    //输出“The 'quote' is &lt;b&gt;bold&lt;/b&gt;”

//中文字符，非 ASCII 字符转换
$str = "中文's Home Page";
echo htmlentities($str);    //输出“&Ouml;&ETH;&Icirc;&Auml;&#039;s Home
                             //在浏览器页面中显示“ÖÐÎÄ's Home Page”，这是不对的
                             Page”

//正确的方法
echo htmlentities($str, ENT_QUOTES, "GB2312");
                             //输出“中文&#039;s Home Page”
                             //在浏览器页面中显示“中文's Home Page”
?>
```

11.6.3 URL 字符串的解析

除了对 HTML 文字进行处理外，在 Web 开发时还要对 URL 地址进行处理。URL 地址可以在浏览器地址栏中看到，它是类似下面形式的字符串。

```
http://usr:pwd@blog.taodoor.com/path/index.php?act=show&id=123&page=1#top
```

1. 解析 URL 字符串

为了能够处理 URL 的信息，可以使用函数 `parse_url()` 对其进行全面分析。此函数将返回一个数组，结果中包含以下部分或者全部内容。

- `scheme`: 协议，如“http”或“https”。
- `host`: 域名。
- `port`: 端口，如“80”。

- user: 认证用户名。
- pass: 认证密码。
- path: 访问路径。
- query: 查询字符串, URL 中 “?” 后的内容, `$_SERVER["QUERY_STRING"]`。
- fragment – 锚, 页面的定位标记, URL 中 “#” 后的内容。

例如, 下面的代码是对前面 URL 字符串的分析。

```
<?php
//完整的 URL 解析
$url_str = " http://usr:pwd@blog.taodoor.com/path/index.php?act=show&id=
123&page=1#top";
$url_array = parse_url($url_str);
print_r($url_array);
?>
```

下面是输出的结果:

```
Array
(
    [scheme] => http
    [host] => blog.taodoor.com
    [user] => usr
    [pass] => pwd
    [path] => /path/index.php
    [query] => act=show&id=123&page=1
    [fragment] => top
)
```

注意: 函数 `parse_url()` 并不能保证给定 URL 的合法性, 它只是将 URL 的各部分分开。`parse_url()` 可接受不完整的 URL, 并尽量将其解析正确。对于相对路径的 URL, 该函数不起作用。

2. 解析查询字符串

URL 的查询字符串 (QUERY_STRING), 通常也是分析的重点。一个查询字符串, 即前面结构数组中的 `query` 元素值, 也可以通过 `$_SERVER["QUERY_STRING"]` 变量获取。

使用 `parse_str()` 函数可以解析 QUERY_STRING 查询字符串, 其函数原型如下所示。

```
void parse_str (string $query_str)
void parse_str (string $query_str, array &$amp;query_arr)
```

`parse_str()` 函数有两种使用方式: 其一是将查询字符串 `$query_str` 中的变量直接转化为 PHP 的同名变量, 其二是将解析后的变量放入指定的 `$query_arr` 数组中。无论使用何种方式, 解析后变量的作用域都将与 `parse_str()` 函数相同。

例如, 在浏览器的地址栏中输入下面的地址:

```
http://localhost/test.php?action=test&str[]=hello&str[]=php&str[]=world"
```

运行下面的程序:

```
<?php
//输出 QUERY_STRING 查询字符串
echo '$_SERVER["QUERY_STRING"] = ';
echo $_SERVER["QUERY_STRING"];
echo "\n";

//第一种方式, 将查询字符串解析为变量
echo "解析为变量: \n";
parse_str($_SERVER["QUERY_STRING"]);
printf("action = %s\n", $action);
```

```

printf("str[0] = %s\n", $str[0]);
printf("str[1] = %s\n", $str[1]);
printf("str[2] = %s\n", $str[2]);

//使用第二中方式，将变量解析到给定的数组中
echo "解析为数组：\n";
parse_str($_SERVER["QUERY_STRING"], $arr);
print_r($arr);
?>

```

执行结果如下：

```

$_SERVER["QUERY_STRING"] = action=test&str[]=hello&str[]=php&str[]=world
解析为变量：
action = test
str[0] = hello
str[1] = php
str[2] = world
解析为数组：
Array(
    [action] => test
    [str] => Array
        (
            [0] => hello
            [1] => php
            [2] => world
        )
)

```

11.6.4 URL 编码处理

一个合法的 URL 包括一些必要的分隔符，如 “/”、“:”、“@”、“.”、“=”、“?”、“&”或“#”等。如果要在 URL 的 QUERY_STRING 查询字符串变量中使用 “&”，或者在用户的密码中使用 “@” 字符，势必会造成一定的混乱。

为了解决这种问题，在对 URL 中的普通字符（非分隔符）编码时，做如下规定：除 “-”、“_”、“.” 以及英文字母和数字外，其他任何字符均将转换为由 “%” 和两个十六进制数字表示的转义字符串。如空格将用 “%20” 表示、“<” 使用 “%3C” 表示等。

可以使用函数 `rawurlencode()` 和 `urlencode()` 进行这种 URL 编码，它们的不同之处在于，`urlencode()` 函数会将空格转换为 “+”。例如下面的程序代码。

```

<?php
//测试 rawurlencode() 函数
$url = 'ftp://www.taodoor.com';
$url .= '/'.rawurlencode("文档");
$url .= '/'.rawurlencode("PHP 教程.doc");           // “PHP” 和 “教程” 之间有个空格

printf('<a href="%s">文件下载</a>', $url);
echo "\n";

//测试 urlencode() 函数
$url = 'ftp://www.taodoor.com';
$url .= '/'.urlencode("文档");
$url .= '/'.urlencode("PHP 教程.doc");           // “PHP” 和 “教程” 之间有个空格

printf('<a href="%s">文件下载</a>', $url);
?>

```

程序的执行结果如下：

```
<a href="ftp://www.taodoor.com/%CE%C4%B5%B5/PHP%20%BD%CC%B3%CC.doc">文件下载</a>
<a href="ftp://www.taodoor.com/%CE%C4%B5%B5/PHP+%BD%CC%B3%CC.doc">文件下载</a>
```

从上面的结果可以看出，函数 `rawurlencode()` 将空格转换为 “%20”，而 `urlencode()` 将其转换为 “+”。此外，PHP 也可以实现 URL 的解码处理。与上述两个编码函数对应的解码函数分别是 `rawurldecode()` 和 `urldecode()`。例如下面的程序。

```
<?php
    echo rawurldecode('%CE%C4%B5%B5');           //输出“文档”
    echo urldecode('PHP+%BD%CC%B3%CC');         //输出“'PHP 教程”
?>
```

11.6.5 查询字符串的构造

使用 `http_build_query()` 函数，可以十分方便地构造 QUERY_STRING 查询字符串。其函数原型如下所示。

```
string http_build_query(array $formdata [, string $numeric_prefix])
```

函数 `http_build_query()` 接受一个 `$formdata` 数组，它可以是简单的一维数组，也可以是多维数组。为了确保在解析 QUERY_STRING 查询字符串时可以获得合法的变量名，应该使用 `$numeric_prefix` 参数。如果 `$formdata` 数组使用了带有数字下标的元素，则 `$numeric_prefix` 会作为数字下标元素的前缀。

例如下面的代码。

```
<?php
    // http_build_query() 使用的例子
    $seek = array(
        'uid'=>123, 'q'=>'关键字', 'c'=>'PHP',
        'other'=>'book'
    );
    $s1 = http_build_query($seek);

    // http_build_query() 使用数字下标的例子
    $seek = array(
        'uid'=>123, 'q'=>'关键字', 'c'=>'PHP',
        'book', 'paper'
    );
    $s2 = http_build_query($seek, 'other_');

    echo "s1.php?$s1\n";
    echo "s15.php?$s2\n";
?>
```

上述程序运行结果如下所示。

```
s1.php?uid=123&q=%B9%D8%BC%FC%D7%D6&c=PHP&other=book
s15.php?uid=123&q=%B9%D8%BC%FC%D7%D6&c=PHP&other_0=book&other_1=paper
```

下面是一段使用多维数组的例子。

```
<?php
    $data = array('user'=>array('name'=>'Lulu, Sun',
                                'age'=>25,
                                'sex'=>'F',
                                'birthday'=>'1982-6-7'),
                  'pastimes'=>array('singing', 'swamming'),
                  '特困生');
    echo http_build_query($data, 'other_');
?>
```

程序的运行结果如下所示。

```
user[name]=Lulu%2C+Sun&user[age]=25&user[sex]=F&user[birthday]=1982-6-7&pastimes[0]=singing&pastimes[1]=swamming&other_0=%CC%D8%C0%A7%C9%FA
```

11.7 小结

在 PHP 的数据结构中，字符串是相对简单的，但是对它的使用却是灵活的。本章主要介绍了与字符串处理相关的方法，涉及字符串的分析、整理、比较、替换、格式化，以及对 HTML 和 URL 字符串的处理。

第 12 章 PHP 与正则表达式的应用

正则表达式（Regular Expression）是一种用来描述字符排列模式的语法规则，它主要用于字符串的模式匹配、查找及替换等运算。正则表达式可以在各种程序语言中使用，如 JavaScript、VB Script、.NET、Java、C#和 PHP 等。

在所有这些程序语言中，正则表达式的语法都是一致的（只是在极个别的地方会有所不同，可以参考相关文档）。PHP 提供了两套正则表达式函数库。一套是 POSIX（Portable Operation System Interface of UNIX）扩展库；另一套是 PCRE（Perl Compatible Regular Expression）扩展库。

本章首先介绍了 POSIX 和 PCRE 扩展库中正则表达式函数的使用，然后通过一系列实例，加深读者对 PHP 中正则表达式应用的理解。

12.1 关于 POSIX 扩展库的正则表达式函数

POSIX 表示可移植运算系统接口（Portable Operating System Interface of UNIX）。POSIX 标准的最初目的是为了提高 UNIX 环境下应用程序的可移植性。POSIX 提供了对正则表达式的支持，PHP 可以使用该标准扩展下的正则表达式。

POSIX 扩展库中主要有以下函数，如 `ereg()`、`eregi()`、`ereg_replace()`、`eregi_replace()`、`split()`、`spliti()`。这些函数可以分为三组，分别用于正则表达式的匹配、替换和拆分运算。下面将分别予以详细介绍。

12.1.1 模式匹配

函数 `ereg()` 和 `eregi()` 是 POSIX 扩展库中正则表达式的匹配函数。它们的函数原型如下。

```
bool ereg (string $pattern, string $string [, array $regs])
bool eregi (string $pattern, string $string [, array $regs])
```

这两个函数都是用来在 `$string` 字符串中寻找与正则表达式 `$pattern` 所匹配的子串。不同的是，函数 `eregi()` 是以忽略大小写的方式进行的，如果匹配成功则返回 `True`。如果给出了 `$regs` 参数，则正则表达式 `$pattern` 中的匹配项将被存储到其中。

下面是关于函数 `ereg()` 的一个应用范例，这段代码可以用来检验浮点数的字符串格式。

```
<?php
//浮点字符串数组
$floats = array(
    "16.45",
    "-16.45",
    "+16.45",
    "2456.90",
    "34skd",
    "216.",
    "2dt6"
);
```

```
//循环遍历数组，进行浮点数的判断
foreach($floats as $num){
    if(ereg("[+-]?[0-9]*\.[0-9]+$", $num)){
        echo "$num is a stand float\n";
    }else{
        echo "$num is not a float number\n";
    }
}
?>
```

注意：如果只是查找一个字符串中是否包含某个子字符串，建议使用 `strstr()`或 `strpos()`函数。它们将比函数 `ereg()`或 `eregi()`的速度更快。

下面是一段关于字符大小写的匹配，使用函数 `ereg()`和 `eregi()`匹配的结果是不同的。

```
<?php
$string = "Hello world!";

//进行区分大小写的匹配
if(ereg("hello", $string)) {
    print "ereg(): 匹配成功! ";
}

//进行忽略大小写的匹配
if(eregi("hello", $string)) {
    print "eregi(): 匹配成功! ";
}
?>
```

执行结果为：

```
eregi(): 匹配成功!
```

以下代码片断将接受一个标准格式（YYYY-MM-DD）的日期字符串，然后再以其他的格式进行显示。

```
<?php
$date = "2007-12-4"; //一个日期字符串

//将匹配的结果放到$reg 参数中
if(ereg("([0-9]{4})-([0-9]{1,2})-([0-9]{1,2})", $date, $regs)) {
    printf("%04d年%02d月%02d日", $regs[1], $regs[2], $regs[3]);
}else{
    print "无效的日期格式: $date";
}
?>
```

输出结果为：

```
2007年12月04日
```

12.1.2 模式替换

函数 `ereg_replace()`和 `eregi_replace()`用于正则表达式的替换，函数 `eregi_replace()`用于非大小写敏感的场所。它们的函数原型如下所示。

```
string ereg_replace (string $pattern, string $replacement, string $string)
string eregi_replace (string $pattern, string $replacement, string $string)
```

函数 `ereg_replace()`在 `$string` 中搜索模式字符串 `$pattern`，并将匹配结果替换为 `$replacement`。当 `$pattern` 中包含模式单元（或子模式）时，`$replacement` 中形如 “\1” 或 “\$1” 的位置将依次被

- ❑ preg_match_all()
- ❑ preg_grep()
- ❑ preg_quote()
- ❑ preg_replace_callback()
- ❑ preg_replace()
- ❑ preg_split()

12.2.1 对正则表达式匹配

函数 `preg_match()` 进行正则表达式的匹配运算。其函数原型如下。

```
int preg_match (string $pattern, string $content [, array $matches])
```

函数 `preg_match()` 在 `$content` 字符串中搜索与给出的正则表达式 `$pattern` 相匹配的内容。如果提供了 `$matches`，则将匹配结果放入其中。`$matches[0]` 将包含与整个模式匹配的文本，`$matches[1]` 将包含第一个捕获的、与括号中的模式单元所匹配的内容，依此类推。该函数只做一次匹配，并最终返回 0 或 1 的匹配结果数。例如下面的代码。

```
<?php
//需要匹配的字符串。date 函数返回当前时间
$content = "Current date and time is 2007-07-30 10:25 pm.";

//使用通常的方法匹配时间
if (preg_match ("/\d{4}-\d{2}-\d{2} \d{2}:\d{2} [ap]m/", $content, $m))
{
    echo "匹配的时间是: " . $m[0] . "\n";
}

//由于时间的模式明显，也可以简单地匹配
if (preg_match ("/([\d-]{10}) ([\d:]{5} [ap]m)/", $content, $m))
{
    echo "当前日期是: " . $m[1] . "\n";
    echo "当前时间是: " . $m[2] . "\n";
}
?>
```

程序的执行结果如下。

```
匹配的时间是: 2007-07-30 10:25 pm
当前日期是: 2007-07-30
当前时间是: 10:25 pm
```

再如，下面的几个简单例子。

```
<?php
//忽略字母大小写
$string = "I Love LULU";
if(preg_match("/i love lulu/i", $string)){
    echo $string;
}

//忽略空白
$string = "ILoveLulu";
if(preg_match("/i love lu lu/ix", $string)){
    echo $string;
}

//匹配到最近的字符串
```



```

$string = "<b>Cool</b> music<hr>Few years ago...";
if(preg_match("/<.*>/", $string, $m)){
    echo $m[0]; //匹配 "<b>Cool</b> music<hr>"
}
if(preg_match("/<.*>/U", $string, $m)){
    echo $m[0]; //匹配 "<b>"
}
?>

```

12.2.2 取得正则表达式的全部匹配

函数 `preg_match_all()` 进行正则表达式的匹配运算，并取得全部匹配结果，其函数原型如下。

```
int preg_match_all (string $pattern, string $content [, array $matches])
```

与 `preg_match()` 函数类似。如果使用了第三个 `$matches` 参数，则将所有可能的匹配结果放入其中。本函数返回整个模式匹配的次数（可能为零），如果出错则返回 `FALSE`。例如，下面是一段将文本中的 URL 链接地址转换为 HTML 的代码。

```

<?php
//多行文字
$str = <<< STRING
    这是一个包含多个 URL 链接地址的多行文字。
    欢迎访问 http://www.taoboor.com
    获取更多信息
STRING;

//匹配一个 URL，直到出现空白为止
preg_match_all("/http:\/\/\/?[^\s]+/i", $text, $links);

//设置页面显示 URL 地址的长度
$max_size = 40;
foreach($links[0] as $link_url)
{
    $len = strlen($link_url); //计算 URL 的长度。如果超过 $max_size 的设置，则缩短。

    if($len > $max_size)
    {
        $link_text = substr($link_url, 0, $max_size). "...";
    } else {
        $link_text = $link_url;
    }

    //生成 HTML 文字
    $text = str_replace("\n", "<br>\n", $text);
    $text = str_replace($link_url, "<a href='$link_url'>$link_text</a>",
$text);
}
?>

```

程序的输出结果如下。

```

这是一个包含多个 URL 链接地址的多行文字。<br>
欢迎访问<a href='http://www.taoboor.com'>http://www.taoboor.com</a><br>
获取更多信息<br>

```

12.2.3 返回与模式匹配的数组单元

函数 `preg_grep()` 用于返回一个与模式匹配的结果数组，其函数原型如下所示。

```
array preg_grep (string $pattern, array $input)
```

该函数返回一个数组，其中包括数组中与给定的\$pattern 模式相匹配的单元。对于输入数组中的每个元素，preg_grep()也是只进行一次匹配。例如下面的代码。

```
<?php
//科目列表
$subjects = array(
    "Mechanical Engineering",
    "Medicine",
    "Social Science",
    "Agriculture",
    "Commercial Science",
    "Politics"
);

$pattern = "/^[a-z]*$/i"; //匹配的模式字符串
$alonewords = preg_grep($pattern, $subjects); //匹配所有仅由有一个单词组成的科目名
print_r($alonewords); //打印结果
?>
```

其执行结果如下。

```
Array
(
    [1] => Medicine
    [3] => Agriculture
    [5] => Politics
)
```

12.2.4 正则表达式的替换

函数 preg_replace()可以完成正则表达式的替换运算，其函数原型如下。

```
mixed preg_replace (mixed $pattern, mixed $replacement, mixed $subject [, int $limit])
```

函数 preg_replace()较函数 ereg_replace()的功能更加强大，其前三个参数均可以使用数组，第四个参数\$limit 可以设置替换的次数，默认为全部替换。例如下面的代码。

```
<?php
//字符串
$string = Name: {Name}<br>\nEmail: {Email}<br>\nAddress: {Address}<br>\n";

//模式
$patterns =array(
    "{Address}/",
    "{Name}/",
    "{Email}/"
);

//替换字符串
$replacements = array (
    "No.5, Wilson St., New York, U.S.A",
    "Thomas Ching",
    "tom@emailaddress.com",
);

//输出模式替换结果
print preg_replace($patterns, $replacements, $string);
?>
```

输出结果如下。

```
Name: Thomas Ching",
Email: tom@emailaddress.com
Address: No.5, Wilson St., New York, U.S.A
```

提示：对于字符串替换而言，`preg_replace()`函数使用了 Perl 兼容正则表达式语法，通常是比函数 `ereg_replace()` 更快的替代方案。如果仅对字符串做简单的替换，则可以使用 `str_replace()` 函数。

12.2.5 正则表达式的拆分

函数 `preg_split()` 与 `split()` 功能一致。例如，下面是一个查找文章中单词数量的代码。

```
<?php
    $seek = array();
    $text = "I have a dream that one day I can make it. So just do it, nothing
is impossible!";

    //将字符串按空白、标点符号拆分（每个标点后也可能跟有空格）
    $words = preg_split("/[.,;!\s']\s*/", $text);
    foreach($words as $val)
    {
        $seek[strtolower($val)] ++;
    }

    echo "共有大约" . count($words) . "个单词。";
    echo "其中共有" . $seek['i'] . "个单词“I”。";
?>
```

注意：`preg_split()` 函数使用了 Perl 兼容正则表达式语法，通常是比 `split()` 函数更快的替代方案。使用正则表达式的方法分割字符串，可以使用更广泛的分隔字符。如果仅用某个特定的字符进行分割，则建议使用 `explode()` 函数，它不调用正则表达式引擎，因此速度是最快的。

12.3 PHP 与正则表达式的综合应用

正则表达式是一套功能强大的字符串处理工具。通过前面的一些代码片段和实例，可以了解 PHP 中正则表达式相关函数的使用。下面，将通过更多的实用技巧和高级实例，进一步展示 PHP 与正则表达式的综合应用。

12.3.1 表单验证

在 Web 程序开发中，对表单数据的验证是十分重要的。例如，某个字段必须被填写、必须是数字、必须是指定的位数，等等。表单验证程序可以确保服务器接收到的数据是经过过滤的、合法的。这也是抵御 Web 攻击、拦截 XSS 漏洞的基本策略。

验证表单数据，通常是对输入数据的格式进行逻辑判断。正则表达式可以十分简便地对字符串格式进行匹配，因此它也经常应用于表单验证程序中。表单验证的项目十分广泛，下面是一些被广泛使用的验证项目。

- 电子邮件地址
- URL 链接地址

- ❑ 日期字符串
- ❑ 电话号码
- ❑ 邮政编码
- ❑ 身份证号码
- ❑ 信用卡号码

这些都是 Web 开发中常见的数据格式。在前面的章节中，已经对它们的正则表达式做过详细的讲解。这里将通过一个实例程序，说明 PHP 是如何实现对表单数据进行验证的。该程序主要分为三部分：验证函数的定义、表单验证程序和生成表单页面的部分。

该程序的代码如下所示。

```
<?php
// ----- 表单验证函数的定义 -----

//中文字符的校验
function check_chinese($str){
    $pattern = "^[".chr(0x01)."-".chr(0xff)."]+$";
    return ereg($pattern,$str);
}

//英文字符的校验
function check_english($str){
    $pattern = "/^[a-z]+$/i";
    return preg_match($pattern, $str);
}

//身份证的校验
function check_idcard($str){
    $pattern15 = "/^\d{8}((0\d)|(1[0-2]))((3[01])|([0-2]\d))\d{3}$/";
    //15位
    $pattern18 = "/^\d{6}((1[89])|(2\d))\d{2}((0\d)|(1[0-2]))((3[01])|([0-2]\d))\d{3}(\d|X)$/"; //18位
    if(preg_match($pattern18, $str) || preg_match($pattern15, $str)){
        return true;
    }else{
        return false;
    }
}

//日期的校验
function check_date($str){
    return preg_match("/^\d{4}-([1-9]|(1[0-2]))-([1-9]|((1|2)\d)|(3(0|1)))$/",
    $str);
}

//电子邮件地址的校验
function check_email($str){
    return preg_match("/^w+([+.]w+)*@w+([-.]w+)*\.\w+([-.]w+)*$/", $str);
}

//URL 链接地址的校验
function check_url($str){
    return preg_match("/^http:\/\/[A-Za-z0-9]+\.[A-Za-z0-9]+[\/=\?%\-&~`@[\]\':;!]*([<>\"'])*$/", $str);
}
```

```

//QQ 号码的校验
function check_qq($str){
    return preg_match("/^[1-9]\d{4,8}$/", $str);
}

//邮政编码的校验
function check_zip($str){
    return preg_match("/^[1-9]\d{5}$/", $str);
}

//手机号码的校验
function check_mobile($str){
    return preg_match("/^((\d{3}\))|(\d{3}\-))?13\d{9}$/", $str);
}

//电话号码的校验
function check_phone($str){
    return preg_match("/^((\d{3}\))|(\d{3}\-))?(0\d{2,3})|0\d{2,3}-)?[1-9]\d{6,7}$/", $str);
}

// ----- 验证的过程 -----
if(!check_chinese($_POST['name'])) $errors[] = "真实姓名的格式错误! ";
if(!check_english($_POST['nick'])) $errors[] = "英文名的格式错误! ";
if($_POST['pass'] <> $_POST['pass2']) $errors[] = "密码的格式错误! ";
if(!check_date($_POST['birthday'])) $errors[] = "生日的格式错误! ";
if(!check_idcard($_POST['idcard'])) $errors[] = "身份证的格式错误! ";
if(!check_email($_POST['email'])) $errors[] = "邮件地址的格式错误! ";
if(!check_url($_POST['homepage'])) $errors[] = "主页地址的格式错误! ";
if(!check_qq($_POST['qq'])) $errors[] = "QQ 号码的格式错误! ";
if(!check_email($_POST['msn'])) $errors[] = "MSN 账号的格式错误! ";
if(!check_phone($_POST['phone'])) $errors[] = "电话号码的格式错误! ";
if(!check_mobile($_POST['mobile'])) $errors[] = "手机号码的格式错误! ";
if(!check_zip($_POST['zip'])) $errors[] = "邮政编码的格式错误! ";

//判断是否有错误信息
if(is_array($errors)){
    echo "<font color=red>";
    foreach($errors as $err){
        echo "<li>$err</li>\n";
    }
    echo "</font>";
}

// ----- 表单页面 -----
?>
<html>
<head>
<title>表单正则表达式校验</title>
<style>
body, td{font:normal 12px Verdana;}
input,textarea,select{font:normal 12px Verdana;color:#000000;border:1px solid #999999;}
td{padding:3px}
</style>
</head>

```

```

<body>
<table align="center" border=1 bordercolor="#333333" style="border-collapse:
collapse;">
  <form action="validator.php" method="post">
    <tr>
      <td>真实姓名: </td><td><input type="text" size="20" name="name" /></td>
      <td>英文名: </td><td><input type="text" size="20" name="nick" /></td>
    </tr>
    <tr>
      <td>密码: </td><td><input type="password" size="20" name="pass" /></td>
      <td>重复密码: </td><td><input type="password" size="20" name="pass2" /></td>
    </tr>
    <tr>
      <td>性别: </td>
      <td><input type="radio" name="sex" value="1" checked /> 男
      <td><input type="radio" name="sex" value="2" /> 女</td>
      <td>生日: </td><td><input type="text" size="20" name="birthday" value=""
/></td>
    </tr>
    <tr>
      <td>身份证: </td><td colspan="3"><input type="text" name="idcard" size="40"
/></td>
    </tr>
    <tr>
      <td>信箱: </td><td><input type="text" size="20" name="email" /></td>
      <td>主页: </td><td><input type="text" size="20" name="homepage" /></td>
    </tr>
    <tr>
      <td>QQ: </td><td><input type="text" size="20" name="qq" /></td>
      <td>MSN: </td><td><input type="text" size="20" name="msn" /></td>
    </tr>
    <tr>
      <td>电话: </td><td><input type="text" size="20" name="phone" /></td>
      <td>手机: </td><td><input type="text" size="20" name="mobile" /></td>
    </tr>
    <tr>
      <td>所在地: </td><td><select name="city">
        <option value="">选择您目前的所在地</option>
        <option value="1">北京</option>
        <option value="2">上海</option>
        <option value="3">重庆</option>
        <option value="4">辽宁</option>
        <option value="5">山东</option>
        <option value="6">浙江</option>
      </select></td>
      <td>邮政编码: </td><td><input type="text" size="20" name="zip" /></td>
    </tr>
    <tr>
      <td>所在地: </td><td colspan="3"><input type="text" name="address" size="40"
/></td>
    </tr>
    <tr>
      <td>爱好: </td>
      <td colspan="3">运动 <input name="favorite[]" value="1" type="checkbox" />
      上网 <input name="favorite[]" value="2" type="checkbox" />
    </tr>
  </form>
</table>

```

```

        听音乐 <input name="favorite[]" value="3" type="checkbox" />
        看书 <input name="favorite[]" value="4" type="checkbox" /></td>
    </tr>
    <tr>
        <td>自我介绍: </td>
        <td colspan="3"><textarea name="description" style="width:80%;height:50px;">
</textarea> </td>
    </tr>
    <tr>
        <td colspan="4" align="center"><input name="submit" type="submit" value="
确定提交" /></td>
    </tr>
</form>
</table>
</body>
</html>

```

12.3.2 UBB 代码

UBB 代码是经常见诸于网络的一种实用技术。UBB 代码使用一种类似于 HTML 风格的书写方式。例如，一个粗体字在 HTML 中使用“粗体”，而在 UBB 代码则是“[b]粗体[/b]”。在输出最终的用户画面时，程序负责将“[b]”这样的符号转变为 HTML 中对应的“”。

这种处理方式，可以避免直接输入 HTML 代码带来的一些不良后果。UBB 代码实际就是一种模式匹配与替换技术。下面就几个简单的标签进行说明。

1. 格式标签

格式标签“[b]”比较简单，其与“[/b]”包围的部分将以粗体显示。将要替换的文字可能是多行，并且对大小写不敏感，应该使用“is”的模式修正符。因此，这个 UBB 代码可以写成“^[b](.*)[/b]is”或者“^[b](.+)[/b]is”，对应的 HTML 标签是“\1”，这意味着模式中第一个括号中的内容将被替换。

下面是格式标签替换的代码。

```

<?php
    $text = <<< TEXT
    [b]粗体文字 1[/b]
    正常文字 [b]粗体文字 2[/b]
TEXT;

    echo "/\[b\](.+)\[\/b\]/is";
    echo "<hr>\n";
    echo preg_replace("/\[b\](.+)\[\/b\]/is", '<b>\1</b>', $text);
?>

```

程序的输出的结果如图 12.1 所示。这显然是有问题的，其中的“正常文字”也被包含进了粗体字标签中。这是因为“*”或“+”是一种贪婪匹配，其将匹配到最后一个可以匹配的字符为止。

修改方法如下，即将其改为“.*?”或“.+?”的非贪婪匹配，也就是仅匹配到最近一个可匹配字符为止。代码如下。

```

echo "/\[b\](..*?)\[\/b\]/is";
echo "<hr>\n";
echo preg_replace("/\[b\](..*?)\[\/b\]/is", '<b>\1</b>', $text);

```

修改后，程序的输出的结果如图 12.2 所示。

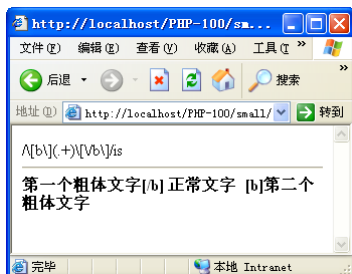


图 12.1 错误的粗体字匹配结果

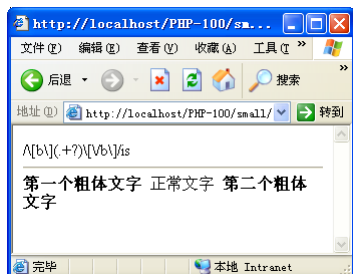


图 12.2 正确的粗体字匹配结果

2. 字体标签

HTML 的“”有字体名称、字体颜色和字体大小三种主要属性。在 UBB 代码中分别使用下面的书写格式。

```
//字体名称
[font=隶书]隶书 ABC[/font]
[font=Times New Roman] Times New Roman[/font]

//字体大小
[size=+1]大 1 号的字体[/size]

//字体颜色，允许使用英文代号，或“#000000”的格式
[color=red]红色的字体[/size]
```

字体名称可以使用“.*?”或“.+?”匹配所有需要替换的内容。但为了说明问题，此处使用一些更精确的匹配模式。对于中文的字体名称，可以使用“[\x7f-\xff]+”匹配；英文字体名称除英文字母外，还可能包含空格和数字，即“[\w]+”，因此，字体名称可以用“[\w\x7f-\xff]+?”表示。

字体大小通常是数字，可能包含“+”、“-”符。可以表示为“[+-]?d{1,2}”。

字体颜色可以用英文名称代号，如“[a-z]{3,}”，也可以使用颜色值（其中，字符“#”并非必需的）的格式，如“#[0-9a-f]{6}”。拼合起来，即“[a-z]{3,}|#[0-9a-f]{6}”。

3. 电子邮件标签

一个电子邮件地址，在 UBB 代码中可以写成如下形式。

```
[email] somebody@email.com[/email] //简单的格式
[email=somebody@email.com]电子邮件[/email] //完整的格式
```

两者的不同之处在于，“完整格式”的邮件地址写在了标签之中、等号“=”之后。下面是这两种格式分别对应的 HTML 代码。

```
<a href="mailto:somebody@email.com">电子邮件</a>
<a href="mailto:somebody@email.com">somebody@email.com</a>
```

和“[b]”的模式类似，使用“.+?”表示一个需要替换的部分。例如：

```
preg_replace("/\[email\](.+?)\[\/email\]/is", '<a href="mailto:\\1">\\1</a>', $text);
preg_replace("/\[email=(.+?)\](.+?)\[\/email\]/is", '<a href="mailto:\\1">\\2</a>', $text);
```

如果要强调电子邮件地址的格式，可以将“.+?”改为“[.\w]+?@[.\w]+?”。“[.\w]+?”表示可以匹配“.”和“\w”格式的字符。“?”的作用依然是非贪婪的匹配。如下面的代码。

```
preg_replace("/\[email\]([.\w]+?@[.\w]+?)\[\/email\]/is", '<a href="mailto:\\1">\\1</a>', $text);
preg_replace("/\[email=([.\w]+?@[.\w]+?)\](.+?)\[\/email\]/is", '<a href="mailto:\\1">\\2</a>', $text);
```


4. UBB 代码

下面给出一个详细的将 UBB 代码转换为 HTML 的代码。其中一部分内容并没有给出具体分析, 但根据前面的讲述, 读者完全可以自行理解。UBBCode.php 的源代码如下所示。

```
<?php
//文件名: UBBCode.php
//功能: 将 UBB 代码转换为 HTML
//输入: 字符串
//输出: 字符串
function converUBB($strCode)
{
    //模式
    $pattern = array(
        //基本样式
        "/\[b\](.+?)\[\/b\]/is",
        "/\[u\](.+?)\[\/u\]/is",
        "/\[i\](.+?)\[\/i\]/is",

        //字体格式
        "/\[font=([\w\x7f-\xff]+?)\](.+?)\[\/font\]/is",
        "/\[color=([a-z]{3,}|#[0-9a-f]{6})\](.+?)\[\/color\]/is",
        "/\[size=([+-]?d{1,2})\](.+?)\[\/size\]/is",

        //电子邮件地址格式
        "/\[email=([a-z]+?@[a-z]+?)\](.+?)\[\/email\]/is",
        "/\[email\]([a-z]+?@[a-z]+?)\[\/email\]/is",

        //URL 链接地址格式
        "/\[url=(.+?)\](.+?)\[\/url\]/is",
        "/\[url\]www\.(.+?)\[\/url\]/is",
        "/\[url\](.+?)\[\/url\]/is",

        //其他常用格式
        "/\[fly\](.+?)\[\/fly\]/is", //移动的字体样式
        "/\[align=(left|center|right)\](.+?)\[\/align\]/is", //字体对齐格式
        "/\[code\](.+?)\[\/code\]/is", //代码格式
    );

    //替换数组
    $replacement = array(
        '<b>\1</b>',
        '<u>\1</u>',
        '<i>\1</i>',
        '<font face="\1">\2</font>',
        '<font color="\1">\2</font>',
        '<font size="\1">\2</font>',
        '<a href="mailto:\1">\2</a>',
        '<a href="mailto:\1">\1</a>',
        '<a href="\1" target=_blank>\2</a>',
        '<a href="http://www.\1" target=_blank>\1</a>',
        '<a href="\1" target=_blank>\1</a>',
        '<marquee width=90% behavior=alternate scrollamount=3>\1</marquee>',
        '<div align=\1>\2</div>',
        '<blockquote><b>代码:</b><hr color=#990000>\1<hr color=#990000></blockquote>',
    );
};
```

```

//进行替换
$string = preg_replace($pattern, $replacement, $strCode);

//返回结果
return $string;
}
?>

```

5. UBB 代码的测试

下面给出了一个段测试自定义 UBB 代码。首先生成一个正文输入表单，文件名为 `ubb.html`，其源代码如下所示。

```

<html>
<head>
<title>UBB 代码测试的输入界面</title>
</head>
<body>
<form action="ubb.php" method="POST">
<p>标题:
    <input type="text" name="title" size="40" value="">
</p>
<p>正文:
    <textarea cols="40" rows="8" name="content"></textarea>
</p>
<p><input type="submit" value="提交"></p>
</form>
</body>
</html>

```

在该表单中输入一些 UBB 代码，如图 12.3 所示。在单击【提交】按钮后，将表单数据提交到 `ubb.php` 文件中处理。该文件的源代码如下所示。

```

<?php
    include 'UBBCode.php';        //引用文件 UBBCode.php
?>
<html>
<head>
<title>UBB 代码测试的输出结果</title>
<style>
h3 { background-color:yellow}
</style>
</head>
<body>
<h3>标题: </h3>
<p><?=htmlspecialchars($_POST['title']) ?></p>
<h3>正文: </h3>
<p><?=nl2br(converUBB($_POST['content'])) ?></p>
</body>
</html>

```

程序 `ubb.php` 首先引用 `UBBCode.php` 文件，载入用户自定义的 `converUBB()` 函数。在接收到提交的“正文”内容后，程序会将 UBB 代码转化为对应的 HTML 代码。程序的运行结果如图 12.4 所示。

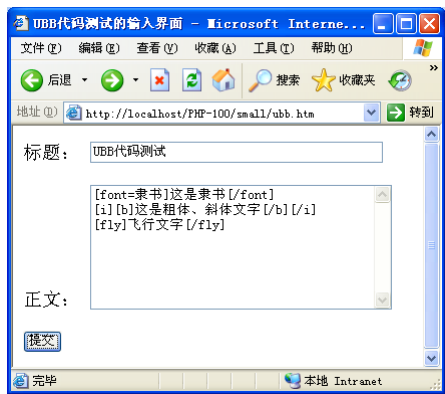


图 12.3 UBB 代码输入的测试文字



图 12.4 UBB 代码的测试结果

12.3.3 分析 Apache 日志文件

通常，为了便于维护，计算机所记录的信息是具有一定规则、但可读性较差的数据格式。需要通过某一方法将其转化为更符合阅读习惯的格式。

例如，图 12.5 显示的是 Apache 服务器的日志文件 access.log。该文件的内容是由计算机自动生成的，不易阅读。

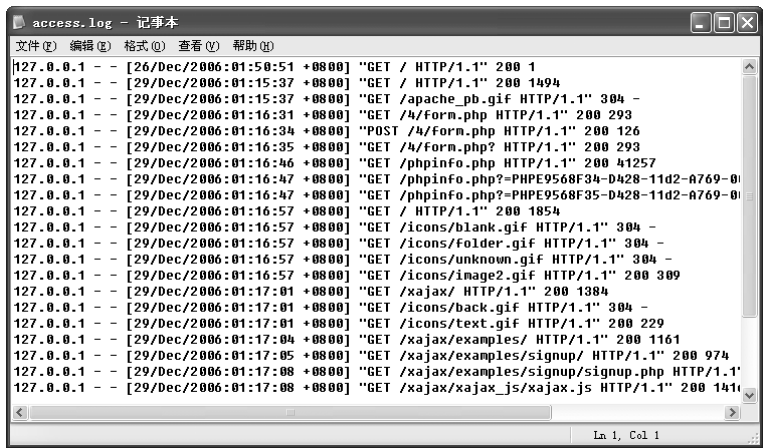


图 12.5 Apache 服务器的日志文件 access.log

从图 12.5 可以看出，access.log 日志文件的内容由多行组成，每行记录都类似于下面的内容。
192.168.0.1 - root [4/Jul/2007:13:27:25 +0800] "GET /code.php HTTP/1.1" 200 120

这是一种典型的 Apache 日志文件格式，要获得这种格式的日志数据，必须在 Apache 的配置文件中 httpd.conf 中进行如下设置。

```
LogFormat "%h %l %u %t \"%r\" %>s %b" common
CustomLog logs/access.log common
```

这里，命令“LogFormat”说明了 access.log 日志文件的存储格式。表 12.1 列出了该格式定义的各项目的含义。命令“CustomLog”定义了该日志文件的存储位置。

表 12.1 access.log 日志文件的存储格式

格式	值	说明
%h	127.0.0.1	客户端的 IP 地址

续表

格式	值	说明
%l	-	访客身份标识。由客户端提供，所以并不可靠
%u	root	HTTP访问用户认证标识，由服务器端决定
%t	[4/Jul/2007:13:27:25 +0800]	脚本执行时的时间
%r	GET /test.php HTTP/1.1	客户端的请求信息，包括请求方法（GET或POST）、被请求的文件或资源、客户端协议
%>s	200	服务器的返回状态。例如： 2xx表示成功； 4xx表示客户端错误； 5xx表示服务器端错误等
%b	120	服务器发送给客户端的数据大小

在了解了数据格式之后，就要分析一下究竟需要格式化哪些数据。因为有些数据并不需要关心。例如每次查询时，access.log 日志将客户端请求的脚本路径记录下来。此外，还包括随之下载的图片、样式表文件等次要信息，这些都是不相干的数据，可以忽略。而且，对于不可靠访客身份标识（%l）等数据也无须考虑。

显然，只要通过判断引号中的被请求文件名，就可以排除大量无用的记录。这里使用正则表达式 “\^[^\.]*(.php|html|html)?\?” 匹配文件名。文件名的前面必定是一个 “/”。方括号 “[^\.]” 表示不包含路径分隔符 “/”，即文件名本身。“\.(php|html|html)” 表明了文件名的后缀，仅限于 PHP 文件或 HTML 文件。最后面的 “\?” 表示文件名中可能存在的参数信息，如存在 “code.php?act=submit” 格式的文件记录。最后将文件名放在引号 “” 中，代码如下所示。

```
<?php
//被请求文件名的匹配
$filename = "\^[^\.]*(.php|html|html)\?";

//将文件名放在引号之内，“.”表示文件名两端的其余信息
$pattern = "\^[^\.]*(.php|html|html)\?";
preg_match($pattern, $logline);
?>
```

对于符合上述条件的记录行，可以使用空白 “\s” 作为分隔符，依次进行匹配。IP 地址使用 “[d.]+” 表示数字和 “.” 的组合，使用方括号匹配时间，即 “[\d.+\?]+”。

引号中的请求信息分为三部分。

- ❑ 第一部分是请求方法，有两种：“POST” 或 “GET”。
- ❑ 第二部分是被请求的文件资源，可以用上面的方法，但实际上，此处也可以看作一个非空白的连续字符串，简化为 “[^\s]+” 完成匹配。
- ❑ 第三部分是客户端协议部分，使用 “HTTPV1\.[0123]” 进行匹配。

下面的代码是 Apache 服务器日志格式化程序的源代码。

```
<html>
<head>
<title>格式化 Apache 服务器日志文件</title>
</head>
<body>
<?php
//将 access.log 日志读入一个数组
$loginfo = file('usr/apache2/logs/access.log');
$i=0;
```

```

?>

<table border=1><tr>
  <th>客户端 IP</th>
  <th>时间</th>
  <th>发送方式</th>
  <th>客户端协议</th>
  <th>请求文件</th>
</tr>

<?php
//遍历日志信息数组
foreach($loginfo as $logline)
{
    //被请求文件名的匹配。只接受后缀为“.php”、“.html”、“.htm”的文件
    if(preg_match("/\".*\[/[^\/] +\.(php|html|htm)\?\?.*\"/i", $logline))
    {
        echo "<tr>\n";

        //进行匹配
        $pattern =
            "([\d.]+\s" //IP 地址
            . "([-w]+\s" //身份标识
            . "\[(.+?)\]\s" //匹配时间
            . "\"(POST|GET)\s([^\s]+\s(HTTP\1\.[0123]))\s\"" //请求查询信息
            . "(\d+)\s(\d+)\s";

        preg_match("/$pattern/ix", $logline, $m); //使用“x”表示忽略空白

        echo " <td>{$m[1]}</td>\n"; //显示 IP 地址

        //重新格式化时间，通过“/”，“:”进行分隔
        $dt = split("[\\/: ]", $m[4]);
        $newtime = strtotime ("{$dt[0]} {$dt[1]} {$dt[2]} {$dt[3]}:{$dt[4]}:{$dt[5]}
{$dt[6]}");
        $newtime = date("[O] Y-m-d H:i:s", $newtime);

        echo " <td align=left>{$newtime}</td>\n"; //显示时间
        echo " <td align=center>{$m[5]}</td>\n"; //显示发送方式
        echo " <td align=center>{$m[7]}</td>\n"; //显示客户端协议
        echo " <td align=left>{$m[6]}</td>\n"; //显示请求文件
        echo "</td>\n";
    }
}
echo "</table>";
?>
</body>
</html>

```

图 12.6 显示了该文件被格式化后的输出效果，通过对比可知，显然大大提高了 Apache 日志的可读性。



客户端IP	时间	发送方式	客户端协议	请求文件
127.0.0.1	[+0800] 2006-12-29 01:16:31	GET	HTTP/1.1	/4/form.php
127.0.0.1	[+0800] 2006-12-29 01:16:34	POST	HTTP/1.1	/4/form.php
127.0.0.1	[+0800] 2006-12-29 01:16:35	GET	HTTP/1.1	/4/form.php?
127.0.0.1	[+0800] 2006-12-29 01:16:46	GET	HTTP/1.1	/phpinfo.php
127.0.0.1	[+0800] 2006-12-29 01:16:47	GET	HTTP/1.1	/phpinfo.php?=PHPE9568F34-D428-11d2-A769-00AA001ACF42
127.0.0.1	[+0800] 2006-12-29 01:16:47	GET	HTTP/1.1	/phpinfo.php?=PHPE9568F35-D428-11d2-A769-00AA001ACF42
127.0.0.1	[+0800] 2006-12-29 01:17:08	GET	HTTP/1.1	/xajax/examples/signup/signup.php
127.0.0.1	[+0800] 2006-12-29 01:17:14	POST	HTTP/1.1	/xajax/examples/signup/signup.server.php
127.0.0.1	[+0800] 2006-12-29 01:17:37	POST	HTTP/1.1	/xajax/examples/signup/signup.server.php
127.0.0.1	[+0800] 2006-12-29 01:18:10	GET	HTTP/1.1	/xajax/examples/multiply/multiply.php
127.0.0.1	[+0800] 2006-12-29 01:18:11	POST	HTTP/1.1	/xajax/examples/multiply/multiply.server.php

图 12.6 Apache 日志格式化后的效果

12.4 小结

正则表达式是一种强大的字符串处理工具。本章首先讲述了正则表达式在 PHP 中的相关函数。然后结合一些综合实例，具体分析了正则表达的具体应用。本章包含了大量的实际例程和相关代码。在掌握了本章的知识后，读者可结合实际加深对正则表达式的理解，并不断提高自己的编程技巧和经验。

第 4 篇 JSP 正则表达式应用

第 13 章 常见的 JSP 中数据处理

JSP (Java Server Page) 技术是 Sun 公司为创建动态 Web 内容而推出的一种技术。其实 JSP 就是一种 HTML 文档。但不同的是，它是由 Java 提供生成动态的内容。因此，与在 Java 中一样，JSP 中的数据也是按照某种标准格式构造的，其数据类型与 Java 相同。在 JSP 中常常会用到对数据的处理，本章将主要介绍 JSP 中常用到的基本的数据处理，主要内容如下。

- JSP 中的常用数据类型
- JSP 中数据类型的转换
- JSP 中字符串数据的处理

13.1 5 种 JSP 中的常用数据类型

JSP 中常用到的数据类型如表 13.1 所示。

表 13.1 常用数据类型

简单数据类型	整数类型 (integer)	byte、short、int、long
	浮点类型 (float)	float、double
	字符类型 (char)	char
	布尔类型 (boolean)	boolean
复合数据类型	类(class)	
	接口 (interface)	
	数组	

本节将陆续介绍简单数据类型与字符串数据类型的使用，并重点介绍以上 JSP 中常用的数据类型及其在 JSP 中的应用。

13.1.1 整数类型及应用

同 Java 一样，JSP 支持 4 个整数类型：字节型 (byte)、短整型 (short)、整型 (int) 和长整型 (long)。

- 整型 (int)：是 32 位的整数数据类型，占有四个字节，因此可存储 $2^{32}=4294967296$ 种不同变化的值。数值范围为-2147483648~2147483647。
- 字节型 (byte)：是有符号的 8 位数据类型，占用一个字节，它可存储 $2^8=256$ 种不同变化的值。通常对于数值较小的变量，可以声明为这种类型。数值范围为-128~127。
- 短整型 (short)：是 16 位的整数数据类型，占有两个字节，因此可存储 $2^{16}=65536$ 种不同变化的值。数值范围为-32768~32767。
- 长整型 (long)：是 64 位的整数数据类型，占有八个字节，因此可存储 2^{64} 种不同变化

的值。数值范围为-9223372036854775808~9223372036854775807。声明长整型数值后要加上“L”或“l”字符。

其中，整型（int）是最常用的数据类型，经常用作控制循环及作为数组的下标。那么，如何在 JSP 中声明一个整型变量呢？下面先来看一个简单的例子。

```
<%@page contentType="text/html; charset=GBK"%>
<html>
<head>
<title>JSP 中整型变量</title>
</head>
<body>
<%!int a = 0;%>
    在 JSP 中声明的整型变量 a 的值初始化为：
<%=a %>
</body>
</html>
```

以上代码中，在<%! %>标签中声明了变量的类型以及变量名。这里首先声明了一个整型变量 *a*，并且将其值初始化为 0。接着通过<%= %>标签，在网页上显示变量 *a* 的当前值。运行结果如图 13.1 所示。

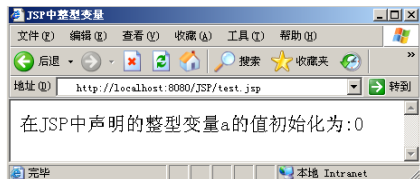


图 13.1 程序运行结果图

JSP 对整型数据的处理同 Java 一样，所不同的只是 JSP 是以网页的形式输出最后数据处理的结果。下面演示将整型变量用作控制循环的例子，在网页中动态输出 0~9 十个整型数值。

```
<%@page contentType="text/html; charset=GBK"%>
<html>
<head>
<title>JSP 中整型变量</title>
</head>
<body>
<%
for(int i=0;i<10;i++){
    %>
        当前 i 的值为：
    <%=i %>
    <br>
    <%
}
%>
</body>
</html>
```

以上代码中，在<% %>标签中的内容是程序的逻辑部分，一些方法的声明与逻辑代码的编写都是在此标签中完成的。上述代码是利用一个 for 循环在网页中从小到大动态地显示从 0 到 9 十个整型数值，运行结果如图 13.2 所示。

字节类型（byte）常用于表示数值较小的数据，由于其范围小，占用系统内存少，因此处理这类数据的速度也相对较快。如果数据数值较小且处于字节型数值覆盖范围内则建议用此类型定

义，以达到节约系统资源、提高处理速度的目的。

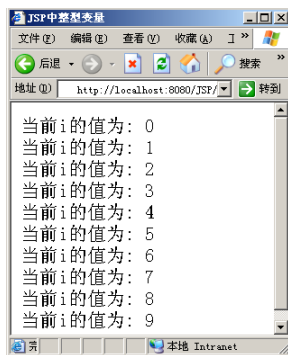


图 13.2 程序运行结果图

相反，长整型（long）常用于表示数值很大的数据，其覆盖数值范围大，但是占用系统内存也大，处理这类数据的速度也相对较慢，因此应当尽量避免将数据定义成长整型，最好根据其所在的数值范围定义为字节型（byte）、整型（int）或是短整型（short）。

几个整数类型的数据在一起进行计算时，要求等号两边的数值类型的最大覆盖范围必须一致。如下面代码所示。

```
<%@ page contentType="text/html; charset=GBK" %>
<html>
<head>
<title>
JSP 中的整数类型
</title>
</head>
<body>
<%!int h=24;           //一天有 24 个小时
    byte m=60;         //一个小时有 60 分钟
    byte s=60;         //一分钟有 60 秒
    short y=365;       //一年有 365 天
    long v=299792458;   //光速的近似值，单位米/秒
    long result=0;      //用于表示一光年的距离
%>
<%
    result=v*s*m*h*y;
%>
一光年的距离相当于<%=result%>米
</body>
</html>
```

在上面的例子中，演示了不同整数类型的数值间的混合计算。需要注意的是，在计算时，等号两边的数值类型的最大覆盖数值范围必须一致。以上面的程序为例，如果将 result 定义为整型（int）或是字节型（byte）都是不可行的，这是由于等式右边的变量 v 是长整型，超过了等式右边变量 result 的表示范围。因此，result 变量必须也定义为长整型（long）。

程序运行结果如图 13.3 所示。



图 13.3 程序运行结果图

13.1.2 浮点类型及应用

浮点类型常用于表示有小数部分的数值，以弥补整数类型在表示数值方面的不足。例如，在数学及物理领域就经常需要用到浮点类型的数值。浮点类型包括两种：**float** 类型与 **double** 类型。

- **float** 类型：由 4 个字节即 32 位组成，**float** 类型范围为 $-3.4\text{e-}38 \sim 3.4\text{e-}38$ ，有效小数位为 7 位。在声明 **float** 类型时，需要在数值后面加 “F” 或 “f”。没有后缀 “f” 的浮点数据则默认为 **double** 类型。
- **double** 类型：精度为 **float** 类型的两倍，也可以称作双精度数，**double** 类型范围为 $-17.7\text{e-}308 \sim 17.7\text{e-}308$ ，有效小数位为 15 位。在声明 **double** 类型时，数值后面可以加后缀 “D” 或 “d”，也可以不加。

需要注意的是，有三种特殊的浮点值用于表示溢出和出错：正无穷大、负无穷大和 NaN（非数值）。例如，某数除以零，得到的结果是正无穷大或是负无穷大；对负数开平方，会得到结果 NaN。

下面的例子演示了在 JSP 中浮点类型数据的应用。

```
<%@ page contentType="text/html; charset=GBK" %>
<html>
<head>
<title>
JSP 中的整数类型
</title>
</head>
<body>
<%!
    float c=3.1415926f;           //圆周率的近似值
    double r=3.33333333333333;   //圆的半径
    double s=0;                  //用于表示圆的面积
%>
<%
    s=c*r*r;
%>
半径为<%=r%>的圆的面积为<%=s%>
</body>
</html>
```

上面的示例演示了计算一个圆的面积，并在网页中输出最后计算结果的过程。代码中用到了两种浮点类型：**float** 类型与 **double** 类型。需要注意的是，在两种类型混合计算时，与整数类型的混合计算一样，等式左边的最大精度类型必须与等式右边相一致。以上面的程序为例，变量 *s* 必须为 **double** 类型，这是因为等式右边的变量 *r* 为 **double** 类型，计算出来的结果也是 **double** 类型，因此 *s* 如果为 **float** 类型，则不足以表示这个 **double** 类型的结果数值。运行结果如图 13.4 所示。



图 13.4 程序运行结果图

13.1.3 字符类型及应用

字符类型用于表示单个字符，是由两个字节即 16 位二进制数表示一个字符，因此字符是无符号数据类型。字符类型的字符可以转换成整数，其整数值范围介于 0~65535 之间。另外，Java 还提供转义字符，以反斜杠（\）开头，将其后的字符转变为另外的含义。表 13.2 列出了 Java 中的转义字符。

表 13.2 转义字符

转义字符	含义
\ddd	表示1~3位八进制数据所表示的字符（ddd）
\uxxxx	表示1~4位十六进制数据所表示的字符（xxxx）
\"	表示双引号字符
\'	表示单引号字符
\\	表示反斜杠字符
\r	表示回车
\n	表示换行
\f	表示换页
\t	表示Tab
\b	表示退格

在 JSP 中，换行、回车等转义字符需要做一定的处理。

```
<%@page contentType="text/html; charset=GBK"%>
<html>
<head>
<title>JSP 中的字符类型</title>
</head>
<body>
<%!
    char c1, c2, c3, c4; //定义三个字符类型变量
    String t;
%>
<%
    c1 = 'A'; //赋值 c1 为'A'
    c2 = '\n'; //赋值 c2 为换行符
    c3 = '\110'; //令 c3 等于八进制 110 的 ASCII 字符，此字符为 H
%>
    字符类型变量 c1 的值为<%=c1%>
<%
    if (c2 == '\n') {                                //判断 c2 是否为换行转义字符
        t = "<br>";                                //如果是则替换为 HTML 格式的换行符输出
    } else                                            //如果不是则输出字符 c2 的值
    } else
%>
    <%=c2%>
    字符类型变量 c3 的值为<%=c3%>
</body>
</html>
```

上述代码中，定义了三个字符类型变量 c1、c2 和 c3，c1 赋值为'A'，c2 赋值为换行符，c3 赋值为八进制 110 的 ASCII 字符。其中，对换行符需要做一定的处理才能正确地在 JSP 网页中输出。首先判断 c2 是否为换行符，如果是则赋值字符串变量 t 为 HTML 格式的换行符“
”，并用 t

替换 c2 输出。运行结果如图 13.5 所示。



图 13.5 程序运行结果图

13.1.4 布尔类型及应用

布尔类型用于表示真与假两种状态。因此，布尔类型的变量只有两个值：`true` 与 `false`。`true` 表示状态为“真”，`false` 表示状态为“假”。布尔类型常常用于控制程序流程，通过关系运算符和逻辑运算符对关系表达式或逻辑表达式进行运算得到布尔值。需要注意的是，布尔值与其他数值之间不可以相互转换。下面请看一个在 JSP 中使用布尔类型数据的简单例子。

```
<%@page contentType="text/html; charset=GBK"%>
<html>
<head>
<title>JSP 中的布尔类型</title>
</head>
<body>
<%!
    boolean pass=false;           //布尔类型变量，初始化为 false，用于标记该学生成绩是否合格
    int score=75;                 //整型变量，代表某学生的分数，初始化为 75
%>
<%
    if(score>=60)                 //如果该学生成绩大于等于 60，则合格
        pass=true;
    else if(score<60)             //如果该学生成绩小于 60，则不合格
        pass=false;
%>
<%
    if(pass){
%>
        该学生的成绩合格！
<%
    }else if(!pass){
%>
        该学生的成绩不合格！
<%
    }
%>
</body>
</html>
```

程序说明：在上面代码中，判断学生的成绩是否大于等于 60 分，如果大于等于，则说明该学生通过了考试，对 `pass` 布尔变量赋值为 `true`，否则赋值为 `false`。并将判断结果输出至浏览器。运行结果如图 13.6 所示。



图 13.6 程序运行结果图

13.1.5 字符串类型及应用

字符串类型的数据在 Java 编程语言中是很重要的部分，在 JSP 中也是一样。字符串类型（String）不属于基本数据类型，String 是 Java 中的类，是复合数据类型。String 类的对象是一种特殊的对象，拥有与其他对象不同的特性。

Java 提供了两个类：String 和 StringBuffer，用于存储和运算字符串。String 和 StringBuffer 分别用于处理不变字符串和可变字符串。首先看下面字符串在 JSP 中应用的一个简单例子。

该例子首先建立一个 HTML 文件 text.html，用于提交一个表单，然后建立一个 JSP 文件 JSPEXample7.jsp，用于接受表单提交的信息，并进行相关处理。text.html 文件代码如下所示。

```
<html>
<head>
<title>JSP 中的字符串数据提交</title>
</head>
<body>
请输入您的个人基本信息并提交：
<form action="JSPEXample7.jsp" method="POST">
输入您的姓名：<input type="text" name="myname" />
<br>
选择您的性别：<select name="sex" size="1">
    <option value="male">男</option>
    <option value="female">女</option>
</select>
<br>
<br>
输入您的邮箱：<input type="text" name="email" />
<br>
<input type="submit" value="提交" />
</form>
</body>
</html>
```

JExample7.jsp 文件代码如下所示。

```
<%@page contentType="text/html; charset=gb2312"%>
<html lang="GB2312">
<head>
<title>JSP 中的字符串数据处理</title>
</head>
<body>
<%!
    String name;
    String sex;
    String E-mail;
%>
%>
<%
    name=request.getParameter("myname");    //获取姓名
    sex=request.getParameter("sex");        //获取性别
    email=request.getParameter("E-mail");    //获取邮箱
%>
<%
    out.print("你的名字是："+name+"<br>");    //显示输出
    if(sex.equals("male"))
        out.print("你的性别是：男<br>");
    else if(sex.equals("female"))
        out.print("你的性别是：女<br>");
```

```

    out.print("你的邮箱是："+email+"<br>");
%>
</body>
</html>

```

上述代码中，提交表单后，由 JSPExample7.jsp 文件负责对接收到的字符串类型的数据进行处理，首先定义三个字符串对象 name、sex、E-mail，分别用于存储姓名、性别和邮箱信息。然后判断用户选择的性别，并输出相应的信息。运行结果如图 13.7 和图 13.8 所示。

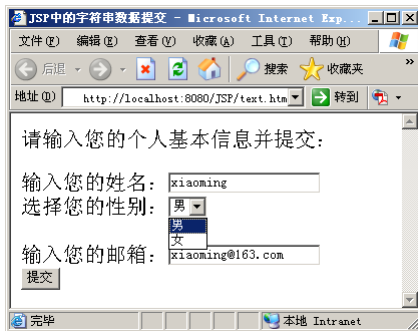


图 13.7 程序运行结果图一

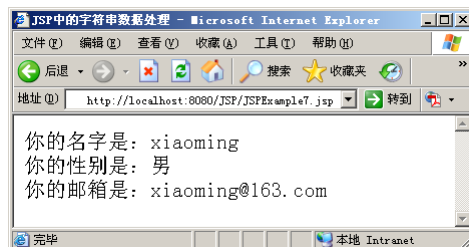


图 13.8 程序运行结果图二

13.2 2 种 JSP 中数据类型的转换

JSP 中数据类型的转换分为两种：自动类型转换和强制类型转换。其中，自动类型转换又称为隐式类型转换，精确度低的数据类型与精确度高的数据类型进行混合运算时，精确度低的数据类型自动转换为精确度高的数据类型。强制类型转换也称为显式类型转换，强制类型转换可以将精确度高的数据类型向精确度低的数据类型转换。

13.2.1 自动类型转换及应用

自动类型转换遵循一定的规则，即总是低级数据向高级数据转换。所谓低级数据与高级数据指的是数据的优先级，基本数据类型的优先级按照由低到高的顺序排列如下：byte、short、char、int、long、float 和 double。

如果要将一个浮点数和一个整型数字相加，则整数会被当作浮点数来对待。需要注意的是，数据类型的变量不会变，只是被转换的值会被用于计算而已。

下面的程序代码给出了一个 JSP 中简单多种数据类型运算的示例，其中，存在类型的混合运算，需要进行类型转换。

```

<%@page contentType="text/html; charset=gb2312"%>
<html lang="GB2312">
<head>
<title>JSP 中的自动类型转换</title>
</head>
<body>
<%!
    char c='A';
    int i=2;
    double d=3.5;
    int ci;           //用于存储字符类型变量 c 与整型变量 i 的和
    double di;        //用于存储整型变量 i 与双精度类型变量 d 的和

```

```

%>
<%
    ci=i+c;
    di=d+i;
%>
字符类型变量 c 的值为: <%=c%>
<br>
整型变量 i 的值为: <%=i%>
<br>
双精度类型变量 d 的值为<%=d%>
<br>
字符类型变量 c 与整型变量 i 的和为: <%=ci%>
<br>
整型变量 i 与双精度类型变量 d 的和为: <%=di%>
</body>
</html>

```

上述代码中, 分别定义了字符类型变量 *c*、整型变量 *i* 和双精度类型变量 *d*。当变量 *c* 与变量 *i* 相加时, 变量 *c* 自动转换为整型(字符 A 的 ASCII 编码为 65, 因此转换为整型数据后数值为 65), 然后与整型变量 *i* 相加, 最后得到整型的数据结果。同样, 当变量 *i* 与变量 *d* 相加时, 变量 *i* 自动转换为双精度类型, 然后与双精度类型变量 *d* 相加, 最后得到双精度类型的数据结果。运行结果如图 13.9 所示。

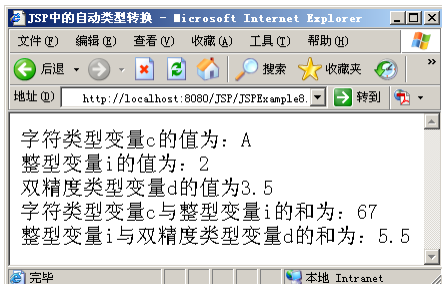


图 13.9 程序运行结果图

13.2.2 强制类型转换及应用

通过强制类型转换可以将高级数据向低级数据转换。创建一个强制转换, 必须将基本类型名用小括号标识出来, 并放在希望转换的值或变量前面。例如下面的代码。

```

double d=2.0;
int i=(int)d; //把 double 型变量 d 强制转换为 int 型

```

需要注意的是, 如果将一个高级数据向一个低级数据转换, 会得到一条警告提示信息, 同时这样做容易引起溢出或导致精度下降, 所以在编程过程中不推荐使用强制类型转换。在 JSP 中, 也是如此。

```

<%@page contentType="text/html; charset=gb2312"%>
<html>
<head>
<title>JSP 中的强制类型转换</title>
</head>
<body>
<%!
    double d1,d2,d3;
    int i;
%>

```

```

<%
    d1=2.5;
    d2=3.7;
    i=(int)d1+(int)d2;
    d3=(double)i/3;
%>
double 类型的两个变量值 d1:<%=d1%>d2:<%=d2%>
<br>
double 类型强制转换为 int 类型并相加的结果为: <%=i%>
<br>
int 类型强制转换为 double 类型并相除的结果为: <%=d3%>
</body>
</html>

```

程序说明：上述代码中，两个 `double` 类型的变量首先强制转换为整型（舍去小数部分），然后相加。第二个运算式中原本不需要强制转换 `int` 类型为 `double` 类型，这里可以自动转换，但是在程序中强制转换也是允许的，这样可以引起程序员的注意。运行结果如图 13.10 所示。

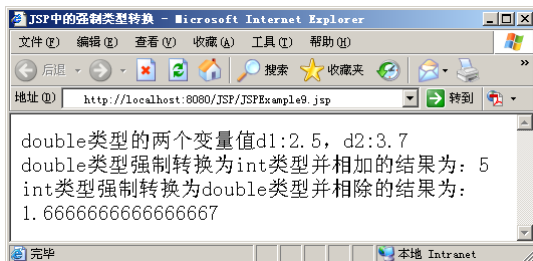


图 13.10 程序运行结果

13.3 7 种 JSP 中字符串数据的处理

前面 13.1.5 节中已经简单介绍了字符串类型，在 JSP 中经常需要对字符串数据进行处理。如字符串与其他类型数据的转换、字符串的分析、字符串查询与替换、字符串数据的整理、字符串的比较和字符串的连接等。下面将对 JSP 中字符串数据的处理做详细介绍。

13.3.1 字符串与其他类型数据的转换

在 JSP 中，整型、浮点型以及字符串等数据可以进行混合运算。运算中，不同类型的数据首先转换为同一类型，然后再进行运算。类型转换主要分为两种：自动类型转换和强制类型转换。

1. 其他数据类型转为字符串类型

对于基本数据类型，可以利用 `String` 类本身提供的静态类方法 `valueOf()`，将逻辑变量、字符、双精度数、浮点数和整数等类型转换为字符串类型。例如下面的代码。

```

int i=1;
String str = String.valueOf(i);           //返回 i 的字符串表示形式

```

对于基本数据类型的包装类（如 `Character`、`Integer`、`Float`、`Double`、`Boolean`、`Short`、`Byte` 和 `Long` 等类）以及其他从 `java.lang.Object` 类派生的类（如 `Exception`、`StringBuffer` 等类），则可以利用 `toString()` 方法将该类转换为字符串。例如下面的代码。

```

int i = 1;
Integer obj = new Integer(i);
String str = obj.toString();              //调用 toString 方法转换为字符串

```


2. 字符串转换为其他数据类型

以基本数据类型的包装类（如 Character、Integer、Float、Double、Boolean、Short、Byte 和 Long）为例，可以利用 `parseXXX()` 方法将字符串转换为相应的数据类型。例如下面的代码。

```
String strInteger = new String("1");
int i= Integer.parseInt(strInteger);
```

下面来看一个在 JSP 中字符串与其他类型数据转换的例子。

```
<%@page contentType="text/html; charset=gb2312"%>
<html>
<head>
<title>JSP 中字符串类型与其他数据类型的转换</title>
</head>
<body>
<%!
    double d1=2.5;
    double d2=3.5;
    Double dObj=new Double(d2);
    int i=0;
    String str1,str2;
    String strInteger="3";
%>
<%
    str1=String.valueOf(d1);           //将双精度类型数据转换为字符串类型
    str2=dObj.toString();              //将双精度类型包装类转换为字符串
    i=Integer.parseInt(strInteger);    //将字符串数据转换为整型数据
%>
    双精度类型数据转换为字符串类型后值为: <%=str1%>
    <br>
    双精度包装类转换为字符串后值为: <%=str2%>
    <br>
    字符串数据转换为整型数据后值为: <%=i%>
</body>
</html>
```

上述代码中，分别以双精度类型数据转换为字符串类型、双精度类型包装类转换为字符串、字符串类型数据转换为整型为例，演示了 JSP 中字符串类型数据与其他类型数据的转换。运行结果如图 13.11 所示。

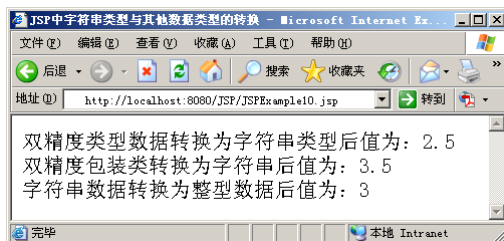


图 13.11 程序运行结果图

13.3.2 字符串的分析

在 JSP 中，常常需要对某个字符串进行分析，如访问某个字符串某个位置的字符、将某个字符串分割成若干子字符串等。在 Java 中，提供了 `charAt` 方法用于访问某个字符串的某个位置的字符，其方法声明如下。

```
public char charAt(int index)
```

其中，`index` 参数是索引值。如果 `index` 参数为负或小于此字符串的长度，则该方法会抛出异常。`charAt` 方法用于在一个特定的位置索引一个字符串，以得到字符串中指定位置的字符。需要注意的是，在字符串中第一个字符的索引是 0，第二个字符的索引是 1，依此类推，最后一个字符的索引是 `length()-1`。

另外，Java 还提供了 `substring` 方法，用于从一个较长的字符串中提取一个子串，该方法返回的是位于 `String` 对象中指定位置的子字符串。其方法声明如下。

```
public String substring (int beginIndex)
public String substring (int beginIndex,int endIndex)
```

第一种声明方法是从字符串中的 `beginIndex` 位置起，从当前字符串中取出剩余的字符作为一个新的字符串返回。该子字符串始于指定索引处的字符，一直到此字符串末尾。

第二种声明方法是从当前字符串中取出一个子串，该子串从 `beginIndex` 位置起至 `endIndex-1` 结束。子串返回的长度为 `endIndex-beginIndex`。需要注意的是，返回的子字符串不包括 `endIndex` 位置的字符。

在 JSP 中，通过以上方法来对字符串数据进行分析。下面请看一个简单的例子，在这个例子中，将对字符串“2007-8-13”进行分析，提取当中的年、月、日数值部分。

```
<%@page contentType="text/html; charset=gb2312"%>
<html>
<head>
<title>JSP 中对字符串数据的分析</title>
</head>
<body>
<%!
    String str="2007-8-13";
    String str1;
    int t=0,n=0;
%>
<%
    for(int i=0;i<str.length();i++){           //遍历字符串 str
        if(str.charAt(i)=='-'){                 //若 i 位置上的字符为“-”则处理
            n+=1;
            str1=str.substring(t,i);           //提取从 t 位置到 i 位置的子串
            %>
            第<%=n%>个数值为: <%=str1%>
            <br>
            <%
                t=i+1;                           //赋值 t 为下一个数值部分的开始位置
            }else if(i==str.length()-1){         //如果 i 为字符串 str 的最后一个字符所在的位置，
            则输出最后一个数值
                n+=1;
                str1=str.substring(t,str.length()); //从字符串 str 中提取 t 位置开始到最后一个
            字符的子串
                %>
                第<%=n%>个数值为: <%=str1%>
                <br>
                <%
                    t=i+1;
                }
            }
        }
    %>
</body>
</html>
```

上述代码中,利用 for 循环遍历字符串“2007-8-13”,当找到字符“-”时则记录该位置,输出该位置前面数值部分的子字符串。当遍历到字符串最后一个字符时,则进行边缘处理,输出最后一个数值部分的子字符串。运行结果如图 13.12 所示。

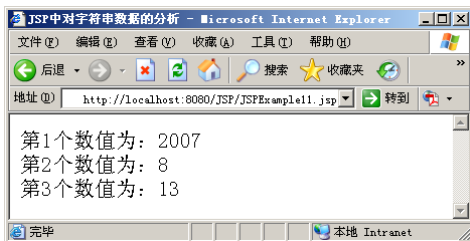


图 13.12 程序运行结果

13.3.3 字符串的查找与替换

在 JSP 中,可以通过 Java 语言提供的 String 类中的 indexOf 方法与 replace 方法对某个字符串进行查找与替换的运算。indexOf 方法用于对某个字符串进行查找的运算,其声明如下:

```
public int indexOf (char ch)
public int indexOf (String str)
public int indexOf (int ch,int fromIndex)
```

前两种方法的声明用于查找当前字符串中某一个特定字符 ch 或子串 str 出现的位置。该方法从头向后查找,如果在字符串中找到字符 ch 或子字符串 str,则返回字符 ch 或子字符串 str 在字符串中第一次出现的位置;如果在整个字符串中没有找到字符 ch 或子字符串 str,则返回-1。

第三种声明方法与前两种方法不同,该方法是从 fromIndex 位置向后查找,返回的仍然是字符 ch 或子字符串 str 在字符串第一次出现的位置。

Replace 方法用于对字符串进行替换的运算,其声明如下:

```
public String replace (char oldChar,char newChar)
```

该方法是用字符 newChar 替换当前字符串中所有的字符 oldChar,并返回一个新的字符串。如果 oldChar 在此 String 对象表示的字符序列中没有出现,则返回对此 String 对象的引用。否则,创建一个新的 String 对象,用来表示与此 String 对象表示的字符序列相等的字符序列,但每个出现的 oldChar 都被一个 newChar 替换除外。

下面的例子中,将由 stringOP.html 文件提交字符串,由 JSPExample12.jsp 文件对接收到的字符串进行处理,将字符串中的空格替换成字符串“---”。stringOP.html 文件代码如下所示。

```
<html>
<head>
<title>JSP 中的字符串数据提交</title>
</head>
<body>
<form action="JSPExample12.jsp" method="POST">
输入字符串: <input type="text" name="theString" />
<input type="submit" value="提交" />
</form>
</body>
</html>
```

Example12.jsp 文件代码如下所示。

```
<%@page contentType="text/html; charset=gb2312"%>
<html>
<head>
```

```

<title>JSP 中对字符串数据的查找与替换</title>
</head>
<body>
<%!
    String str;
    String str1="---";                //待替换字符串
    int t=0,i=0,begin=0;
%>
<%
    str=request.getParameter("theString");    //接收从用户端发来的字符串数据
    while(t!=-1){                             //判断是否已查找完字符串的所有空格字符
        i++;
        t=str.indexOf(' ',begin);             //从 begin 位置开始查找空格字符,返回位置 t
        if(t!=-1)
            out.print("第"+i+"次出现空格的位置为: "+t+"<br>");
        begin=t+1;                            //重新设置 begin 位置
    }
    str=str.replace(" ",str1);                //替换字符串中的空格字符
%>
替换后的字符串为: <%=str%>
</body>
</html>

```

上述代码中,首先利用 `indexOf` 方法查找字符串中的空格符位置并输出,然后利用 `replace` 方法替换字符串中的空格字符并输出。这里需要注意的是,字符串中空格符的位置从 0 开始计算。运行结果如图 13.13 和图 13.14 所示。



图 13.13 程序运行结果图一

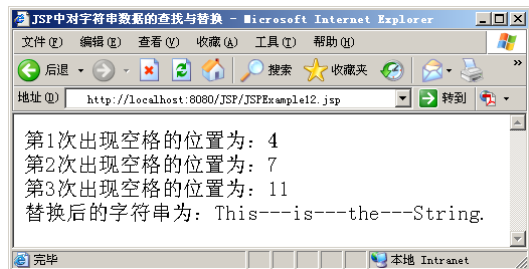


图 13.14 程序运行结果图二

13.3.4 字符串数据的整理

在 JSP 中,常常需要对接收到的字符串进行预整理,如消除字符串中的多余空格部分、将字符串中的所有大写字母转换为小写、反转字符串等。下面的例子中,将通过 Java 语言中 `String` 类的 `toLowerCase()`、`toUpperCase()` 和 `trim()` 等方法实现对字符串数据的整理。

```

<%@page contentType="text/html; charset=gb2312"%>
<html>
<head>
<title>JSP 中对字符串数据的整理</title>
</head>
<body>
<%!
    String str="          This is the String.";
%>
<%
    out.print("原字符串为: "+str+"<br>");
    str=str.toLowerCase();                //将字符串中的大写字母替换为小写字母

```

```

out.print("大写字母替换为小写字母后为: "+str+"<br>");
str=str.trim(); //去除字符串中的多余空格
out.print("去除字符串中的多余空格后为: "+str+"<br>");
StringBuffer strb=new StringBuffer(str); //反转字符串
str=strb.reverse().toString();
out.print("反转字符串后为: "+str+"<br>");
%>
</body>
</html>

```

上述代码中, 分别实现了 JSP 中字符串大小写的替换, 字符串中多余空格的去除以及字符串的反转。其中字符串的反转需要建立一个 `StringBuffer` 对象, 然后通过调用 `StringBuffer` 对象的 `reverse()` 方法反转字符串并输出反转后的字符串。运行结果如图 13.15 所示。

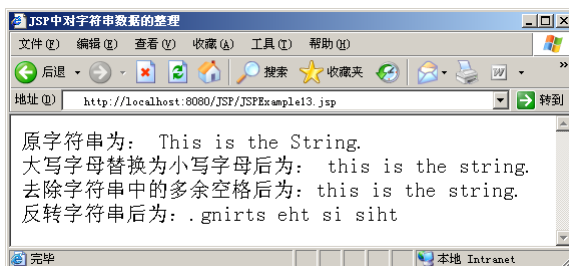


图 13.15 程序运行结果图

13.3.5 字符串的比较

在 JSP 中, 常常需要判断两个字符串是否相等。在 Java 语言中, `String` 类提供了 `equals` 方法, 用于比较两个字符串对象的内容, 返回的是一个布尔类型的值, 如果两个字符串的内容相等则返回 `true`, 否则返回 `false`。例如下面的代码。

```

String str1=newString("hello");
String str2=newString("hello");
boolean b=str1.equals(str2);

```

下面的例子中, 将建立两个文件 `login.html` 和 `JSPEExample14.jsp`, 模拟实现用户登录验证的功能。由 `logon.html` 文件负责提交用户的基本信息 (用户名、密码), 由 `JSPEExample14.jsp` 文件负责对接收到的用户信息进行验证并返回结果。`login.html` 文件代码如下所示。

```

<html>
<head>
<title>JSP 中的字符串数据提交</title>
</head>
<body>
<form action="JSPEExample14.jsp" method="POST">
输入用户名: <input type="text" name="userName" />
<br>
请输入密码: <input type="password" name="userPassWord" />
<br>
<input type="submit" value="提交" />
</form>
</body>
</html>

```

`Example14.jsp` 文件代码如下所示。

```

<%@page contentType="text/html; charset=gbk"%>
<html>

```

```

<head>
<title>JSP 中对字符串数据的整理</title>
</head>
<body>
<%!
    String name,password;
    String SQLResultName="xiaoming"; //假设数据库中存在用户名"xiaoming"
    String SQLResultPassWord="111"; //假设数据库中用户名"xiaoming"对应的密码为"111"
%>
<%
    name=request.getParameter("userName"); //获取用户输入的用户名
    password=request.getParameter("userPassWord"); //获取用户输入的密码
    if((name.equals(SQLResultName)) && (password.equals(SQLResultPassWord))) {
        //用户名、密码验证
    }
    %>
        登录成功! <%=name%>, 欢迎你登录本网站!
    <%
    }else{
    %>
        登录失败! 密码错误或是该用户不存在!
        <br>
        请<a href="logon.html">重新登录</a>
    <%
    }
    %>
</body>
</html>

```

上述代码中，用户提交用户名和密码后，由 JSPExample14.jsp 文件对字符串进行比较验证，这里假设数据库中存在用户名“xiaoming”以及对应的密码“111”。如果输入的用户名、密码与数据库中的用户名、密码的内容相同，则返回登录成功信息，否则登录失败，返回登录页面。运行结果如图 13.16 和图 13.17 所示。

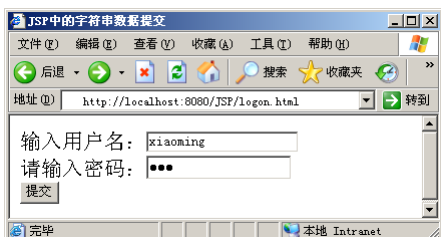


图 13.16 程序运行结果图一

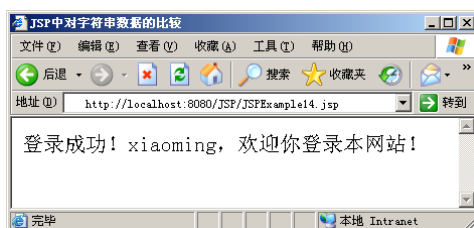


图 13.17 程序运行结果图二

13.3.6 字符串的连接

Java 中，字符串的连接有三种方法。

- 使用 “+” 运算符。
- 使用 String.concat() 方法。
- 使用 StringBuffer.append() 方法。

在 JSP 中也是一样，下面的例子中将分别演示三种方法的应用。

```

<%@page contentType="text/html; charset=gbk"%>
<html>
<head>
<title>JSP 中对字符串数据的连接</title>

```

```

</head>
<body>
<%!
    String str1,str2,str3;
%>
<%
    str1="字符串 1";                //定义三个字符串
    str2="字符串 2";
    str3="字符串 3";
    StringBuffer strb=new StringBuffer(str3);    //定义 StringBuffer 对象,赋值 str3
    out.print("str1 的 值 为 :"+str1+";<br>str2 的 值 为 :"+str2+";<br>str3 的 值
为:"+str3+"<br>");
    str1=str1+str2;                //通过+连接符连接字符串
    out.print("str1+str2 的结果为:<br>"+str1+"<br>");
    str3=str1.concat(str3);        //通过 concat 方法连接字符串
    out.print("将 str3 连接到 str1 与 str2 的末尾的结果为:<br>"+str3+"<br>");
    strb.append(str1);             //通过 append 方法连接字符串
    out.print("将 str1 与 str2 连接到 str3 的末尾的结果为:<br>"+strb.toString()+"<br>");
%>
</body>
</html>

```

上述代码中,分别演示了三种字符串连接方法在 JSP 中的应用。三种方法各有各的特点,在编写程序过程中,应当根据具体情况使用不同的方法,最终达到使程序高效运行的目的。运行结果如图 13.18 所示。

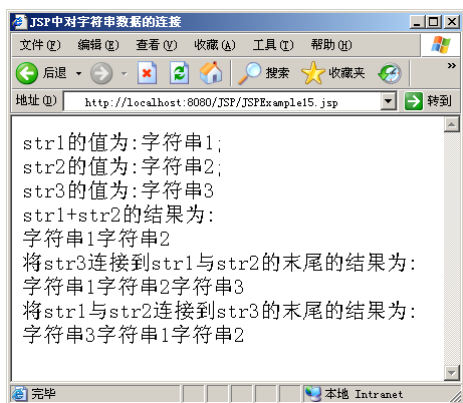


图 13.18 程序运行结果图

13.3.7 字符串的格式化

String 类提供了 format()方法,用于格式化字符串。下面请看一个简单的例子,在例子中,将把字符串格式化为“姓名,年龄,性别”这种形式的格式。

```

<%@page contentType="text/html; charset=gbk" import="java.util.*;" %>
<html>
<head>
<title>JSP 中对字符串的格式化</title>
</head>
<body>
<%!
    Object[] data = new Object[3];                //对象数组,大小为 3
%>

```

```
<%  
    data[0] = "小明";           //姓名  
    data[1] = Integer.valueOf(23); //年龄  
    data[2] = "男";           //性别  
    String dataString = String.format("姓名:%s,年龄:%d,性别:%s",    //格式化字符串  
        (Object[]) data);  
    out.print(dataString);  
%>  
</body>  
</html>
```

上述代码中，通过 `String` 类的 `format` 方法格式化字符串为“姓名，年龄，性别”。其中，`%s` 用于指定返回调用“`对象.toString()`”得到的结果，`%d` 用于指定返回被格式化为十进制整数的结果。程序中，分别将 `Object` 对象数组中的 3 个对象转换为相应的格式去替换“`%s`”、“`%d`”。运行结果如图 13.19 所示。

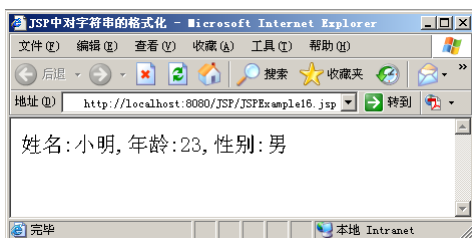


图 13.19 程序运行结果图

13.4 小结

本章详细介绍了 JSP 中的常用数据类型、JSP 中数据类型的转换，以及 JSP 中对字符串数据的相关处理。在本章开始，首先通过具体的应用实例，详细讲解了 Java 中的简单类型以及字符串类型在 JSP 中的应用。在此基础上，介绍了 JSP 中不同数据类型之间的转换，包括自动类型转换与强制类型转换。在最后一节中重点讨论了 JSP 中对字符串数据的处理，并给出相关示例供读者学习。

第 14 章 常见的 JSP 中正则表达式

用户在编写网页中的 JSP 程序时，经常会查找符合某种规则的字符，或者判断字符串参数是否符合某种规则，如输入的邮件地址是否合法等问题。这种情况下，就需要用正则表达式来处理了。正则表达式是可以用于描述并记录文本的规则。

14.1 2 种 JSP 中的正则表达式函数

在 JSP 中使用正则表达式，需要引入 Java 中的 `java.util.regex` 包，它包含了与正则表达式相关的 `Pattern` 类和 `Matcher` 类等。`Pattern` 类表示以字符串形式指定的正则表达式，而 `Matcher` 类的实例用于匹配字符序列与给定模式。

一个正则表达式编译成一个 `Pattern` 类的对象，这个 `Pattern` 对象将会使用 `Pattern` 类的方法 `matcher()` 来产生一个 `Matcher` 对象，接下来就可以使用该 `Matcher` 实例编译的正则表达式对目标字符串进行匹配工作。需要注意的是，多个 `Matcher` 对象可以共用一个 `Pattern` 对象。

14.1.1 `Pattern` 类

`Pattern` 类以字符串的形式指定正则表达式的编译表示形式。`Pattern` 类没有构造方法，所以使用类方法 `compile` 生成一个 `Pattern` 类对象，具体如下。

- ❑ `static Pattern compile(String regex)` 方法：将给定的正则表达式 `regex` 编译，并将其赋值给一个 `Pattern` 对象。
- ❑ `static Pattern compile(String regex, int flags)` 方法：该方法与上面方法的区别是增加了参数 `flag`。`flag` 参数可以是 `CASE_INSENSITIVE`、`MULTILINE`、`DOTALL`、`UNICODE_CASE`、`CANON_EQ`、`UNIX_LINES`、`LITERAL` 和 `COMMENTS`。`flag` 参数能用来改变处理正则表达式的方式，如 `MULTILINE` 表示更改了 `^` 和 `$` 的含义，使其分别表示任意一行的行首和行尾匹配，而不仅仅是在整个字符串的开头和结尾匹配。

`Pattern` 类还提供了其他方法供用户使用，其具体描述及说明如下。

- ❑ `public int flags()` 方法：返回当前 `Pattern` 模式类的匹配 `flag` 参数。
- ❑ `public Matcher matcher(CharSequence input)` 方法：使用指定的参数 `input` 生成一个匹配器 `Matcher` 对象。
- ❑ `public static boolean matches(String regex, CharSequence input)` 方法：利用给定的正则表达式对参数指定的 `input` 字符串进行匹配，匹配模式为该正则表达式的模式。
- ❑ `public String pattern()` 方法：返回该 `Pattern` 对象所编译的正则表达式的字符串形式结果。
- ❑ `public String[] split(CharSequence input)` 方法：将参数字符串 `input` 按照 `Pattern` 里所包含的正则表达式进行分割，返回结果为一个字符串组。
- ❑ `public String[] split(CharSequence input, int limit)` 方法：该方法作用与 `public String[] split(CharSequence input)` 方法一样。不同的是增加了参数 `limit`，`limit` 的作用是限定划分

的个数，即如果 limit 为 3，则只能将指定的字符串分为 3 段。

□ public String toString()方法：返回该匹配模式的字符串表示形式。

```
<%@ page language="java" import="java.util.regex.*" contentType="text/html;
charset=gb2312"%>
<!DOCTYPE HTML PUBLIC "-//w3c//dtd html 4.0 transitional//en">
<html>
<head>
<title>1 JSP</title>
</head>
<body bgcolor="#FFFFFF">

<%
Pattern p = Pattern.compile("[/]+");
//用 Pattern 的 split() 方法把字符串按"/"分割
String[] result = p.split("本书全面介绍了标准的基本知识，"+
    "/涵盖了从正则表达式到高性能 IO 这样的特性，" +
    "/从中能够读到关于面向对象的 Java 的经典阐述." +
    "/keyword:java");
for(int i=0;i<result.length;i++)
{
    String temp;
    temp=result[i];
    out.print(result[i]+"<br>");
}

%>

</body>
</html>
```

程序中利用正则表达式[/]+产生 Pattern 类对象，该表达式表示匹配一个或者多个符号/；接着使用方法 split 将其进行分隔，并将结果输出至浏览器。程序运行结果如图 14.1 所示。

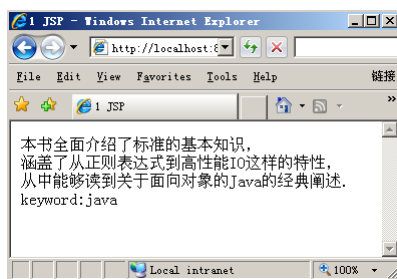


图 14.1 程序运行结果

还可以利用 public String[] split(CharSequence input, int limit)中的 limit 限制分隔的段数。示例代码如下所示。

```
<%@ page language="java" import="java.util.regex.*" contentType="text/html;
charset=gb2312"%>
<!DOCTYPE HTML PUBLIC "-//w3c//dtd html 4.0 transitional//en">
<html>
<head>
<title>2 JSP</title>
</head>
```

```

<body bgcolor="#FFFFFF">

<%
Pattern p = Pattern.compile("[/]+");
//      用 Pattern 的 split() 方法把字符串按 "/" 分割
String[] result = p.split("本书全面介绍了标准的基本知识, "+
    "/涵盖了从正则表达式到高性能 IO 这样的特性,      "+
    "/从中能够读到关于面向对象的 Java 的经典阐述." +
    "/keyword:java",2);
for(int i=0;i<result.length;i++)
{
    out.print(result[i]+"<br>");

}

%>

</body>
</html>

```

与上一个示例唯一不同的是, `split` 方法中的参数 `limit` 限定了分隔的个数不能超过两个。程序运行结果如图 14.2 所示。

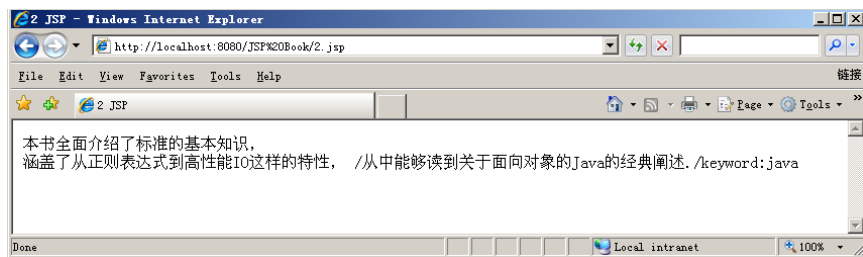


图 14.2 程序运行结果

14.1.2 Matcher 类

`Matcher` 类对象由 `Pattern` 类的方法 `matcher(CharSequence input)` 生成, 该方法描述及作用如下。

- `public Matcher matcher(CharSequence input)` 方法: 使用指定的参数 `input` 生成一个匹配器对象。通过调用这个方法可以产生相应的 `Matcher` 对象。接着就可以利用这个生成的 `matcher` 对象对指定的字符串进行匹配运算, 其中定义如下所示相关的方法。
- `public Matcher appendReplacement(StringBuffer sb, String replacement)` 方法: 将当前匹配的子字符串替换为指定的字符串, 并且将替换后的子字符串, 以及该替换的子字符串之前到上个匹配子字符串位置之间的字符串段添加到一个 `StringBuffer` 对象里。
- `public StringBuffer appendTail(StringBuffer sb)` 方法: 将最后一次匹配后剩余的字符串添加到一个 `StringBuffer` 对象里。
- `public int end()` 方法: 返回当前匹配的子字符串的最后一个字符在原来的字符串中的索引位置。
- `public boolean find()` 方法: 该方法用于查找输入字符串中与该模式匹配的下一个子序列。此方法从匹配器的开头开始。匹配器的开头由上次匹配结果决定, 是匹配成功后的第一个字符。如果匹配器被重置, 则从匹配字符串开头开始。

- ❑ `public boolean find(int start)`方法：重新设置匹配器，并且从目标字符串的指定位置开始查找子字符串。
- ❑ `public String group()`方法：返回当前查找而获得的与组匹配的所有子字符串内容。
- ❑ `public String group(int group)`方法：返回当前匹配的与指定的组匹配的子字符串内容。
- ❑ `public int groupCount()`方法：返回当前查找匹配的组的数量。
- ❑ `public boolean lookingAt()`方法：检测目标字符串是否以匹配的子字符串开始。
- ❑ `public boolean matches()`方法：判断整个字符串是否匹配，如果匹配则返回 `true`，否则为 `false`。
- ❑ `public Pattern pattern()`方法：返回该匹配器对象的现有匹配模式，也就是对应的 `Pattern` 对象。
- ❑ `public String replaceAll(String replacement)`方法：将目标字符串中与现有 `pattern` 模式相匹配的子字符串全部替换为指定的字符串。
- ❑ `public String replaceFirst(String replacement)`方法：将目标字符串第一个与现有 `pattern` 模式相匹配的子字符串替换为指定的字符串。
- ❑ `public Matcher reset()`方法：重新设置该匹配器对象。
- ❑ `public Matcher reset(CharSequence input)`方法：重新设置该匹配器 `matcher` 对象，并且指定一个新的目标字符串参数作为目标字符串。
- ❑ `public int start()`方法：返回当前查找所获得的子字符串的开始字符在原目标字符串中的位置。
- ❑ `public int start(int group)`方法：返回当前查找所获得的和指定组匹配的子字符串的第一个字符在原目标字符串中的位置。

以下代码中，首先新建一个模式匹配器 `p`，将给定的正则表达式“Xinwei”编译到模式中；接着结合指定字符串产生 `matcher` 的实例对象，并利用 `matcher` 的实例对象查找匹配的字符串；然后使用 `appendReplacement` 将其加入 `StringBuffer` 对象中；最后将所有匹配后字符加入，并输出至浏览器。程序运行结果如图 14.3 所示。

```
<%@ page language="java" import="java.util.regex.*" contentType="text/html;
charset=gb2312"%>
<!DOCTYPE HTML PUBLIC "-//w3c//dtd html 4.0 transitional//en">
<html>
<head>
<title>3 JSP</title>
</head>
<body bgcolor="#FFFFFF">

<%
    Pattern p = Pattern.compile("Xinwei");
    // 用 Pattern 类的 matcher() 方法生成一个 Matcher 对象
    Matcher m = p
        .matcher("oh!Xinwei Chen and Xinwei Chan are both working in Xinwei
Chen's XinweiSoftShop company");
    StringBuffer sb = new StringBuffer();
    int i = 0;
    // 使用 find() 方法查找第一个匹配的对象
    boolean result = m.find();
    // 使用循环将句子里所有的 Xinwei 找出并替换为 panglina，再将内容加到 sb 对象中
    while (result) {
        i++;
```

```

        m.appendReplacement(sb, "panglina"); //将"panglina"加入到 sb 对象中
        out.print("第" + i + "次匹配后 sb 的内容是: " + sb);
        out.print("<br>");
        // 继续查找下一个匹配对象
        result = m.find();
    }
    // 最后调用 appendTail() 方法将最后一次匹配后的剩余字符串加到 sb 对象中;
    m.appendTail(sb);
    out.print("调用 m.appendTail(sb) 后 sb 的最终内容是: " + sb.toString());
%>
</body>
</html>

```

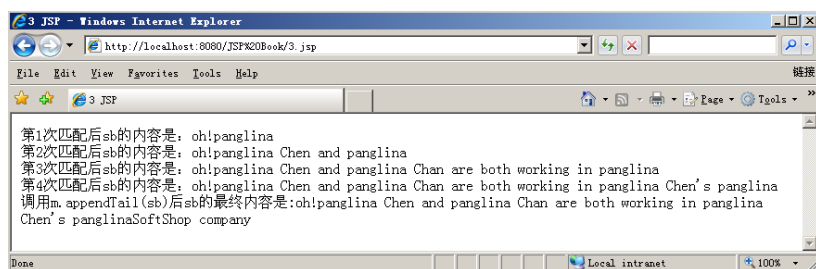


图 14.3 程序运行结果

以下代码中, 利用正则表达式 `(ca)(t)` 创建一个模式匹配器 `p`, 并将给定的正则表达式 `"(ca)(t)"` 编译到模式中。利用 `Matcher` 类的 `groupCount` 方法及 `group` 方法, 计算此匹配器模式中的捕获组数, 并利用 `group` 方法返回在以前匹配运算期间由给定组捕获的输入子序列。

```

<%@ page language="java" import="java.util.regex.*" contentType="text/html;
charset=gb2312"%>
<!DOCTYPE HTML PUBLIC "-//w3c//dtd html 4.0 transitional//en">
<html>
<head>
<title>4 JSP</title>
</head>
<body bgcolor="#FFFFFF">

<%
    Pattern p = Pattern.compile("(ca)(t)"); //创建一个模式匹配器
    //待匹配字符串"one cat,two cats in the yard"
    Matcher m = p.matcher("one cat,two cats in the yard");
    StringBuffer sb = new StringBuffer();
    boolean result = m.find(); //查找, 返回查找是否成功
    out.print("该次查找获得匹配组的数量为: " + m.groupCount());
    out.print("<br>");
    for (int i = 1; i <= m.groupCount(); i++) {
        out.print("第" + i + "组的子串内容为: " + m.group(i));
        out.print("<br>");
    }
%>

</body>
</html>

```

程序运行结果如图 14.4 所示。

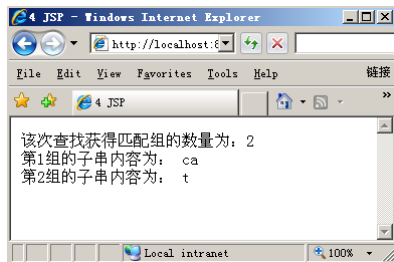


图 14.4 程序运行结果

14.1.3 正则表达式常用的四种功能

通常正则表达式都用于以下四种场合：查找、获取、分段和替换。

1. 查找

事实上，查找功能是一种非常实用的功能，使用 `Matcher` 类的 `find` 方法即可实现查找功能。

例如下面的代码。

```
<%@ page language="java" import="java.util.regex.*" contentType="text/html;
charset=gb2312"%>
<!DOCTYPE HTML PUBLIC "-//w3c//dtd html 4.0 transitional//en">
<html>
<head>
<title>5 JSP</title>
</head>
<body bgcolor="#FFFFFF">

<%
String str="abc efg ABC";

String regEx="a|f"; //表示 a 或 f

Pattern p=Pattern.compile(regEx);

Matcher m=p.matcher(str);

try{
    boolean rs=m.find();
    out.print("在原字符串中是否查找到 a 或者 f? "+rs);
    out.print("<br>");
    }
    catch(Exception e){
        e.printStackTrace();
    }finally{
        out.print("程序运行结束");
    }

%>

</body>
</html>
```

上述代码中首先创建了模式匹配器 `p`，然后利用 `matcher` 对象的 `find` 方法进行查找。如果 `str`

中有 `regEx`, 则 `rs` 为 `true`, 否则为 `false`。如果想在查找时忽略大小写, 则使用代码 `Pattern p=Pattern.compile(regEx,Pattern.CASE_INSENSITIVE)`, 即使用 `flag` 参数指定模式匹配器的匹配模式。程序运行结果如图 14.5 所示。

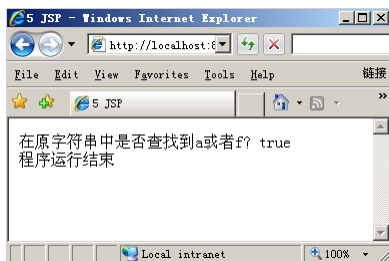


图 14.5 程序运行结果

2. 获取

在正则表达式中可以利用 `group` 方法提取具体某个匹配结果。

```
<%@ page language="java" import="java.util.regex.*" contentType="text/html;
charset=gb2312"%>
<!DOCTYPE HTML PUBLIC "-//w3c//dtd html 4.0 transitional//en">
<html>
<head>
<title>6 JSP</title>
</head>
<body bgcolor="#FFFFFF">

<%
String regEx="(\\w+) (\\.) (\\w+)$";

String str="c:\\dir1\\dir2\\name.txt";

Pattern p=Pattern.compile(regEx);

Matcher m=p.matcher(str);

m.find();

for(int i=1;i<=m.groupCount();i++){

out.print(m.group(i));

}

%>

</body>
</html>
```

程序中利用了 `groupCount` 和 `group` 方法提取具体某个匹配分组的结果。这里读者需要注意循环变量 `i` 的范围设置, 读者可以尝试将其进行变化, 如从 0 开始循环, 看看输出结果会发生怎样的变化。多多动手, 会对 `Pattern` 类和 `Matcher` 类的各个方法理解得更加深入。程序运行结果如图 14.6 所示。

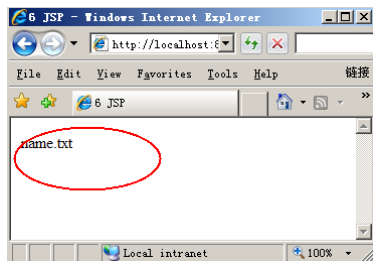


图 14.6 程序运行结果

3. 分隔

前面的章节讲过，使用 `split` 方法可以将指定的字符串使用指定的正则表达式进行分隔。例如下面的代码。

```
<%@ page language="java" import="java.util.regex.*" contentType="text/html;
charset=gb2312"%>
<!DOCTYPE HTML PUBLIC "-//w3c//dtd html 4.0 transitional//en">
<html>
<head>
<title>7 JSP</title>
</head>
<body bgcolor="#FFFFFF">

<%
String regEx=":::";

Pattern p=Pattern.compile(regEx);

String[] r1=p.split("xd::abc::cde");

//执行后，r 就是{"xd","abc","cde"}，其实分割时还有更简单的方法：

String str="xd::abc::cde";

String[] r2=str.split("::");
for(int i=0;i<r1.length;i++)
{
out.print(r1[i]+" ");
}
out.print("<br>");
for(int i=0;i<r2.length;i++)
{
out.print(r2[i]+" ");
}

%>

</body>
</html>
```

程序中给出了 `split` 方法的两种使用方式，一种是通过 `Pattern` 类和 `Matcher` 类，另一种是直接使用 `String` 类提供的该方法。注意，两种方法在相同的正则表达式下，输出结果一模一样。程序运行结果如图 14.7 所示。

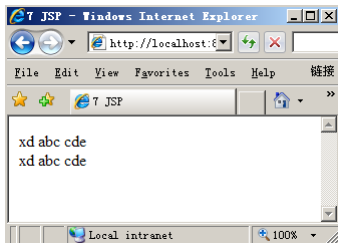


图 14.7 程序运行结果

4. 替换

在字符串的运算中，另一种常见的运算就是替换，如果一篇文章中出现了某个错别字，就可以使用替换运算。**Matcher** 类提供了非常易用的方法 **replaceAll**，可将指定的正则表达式匹配的字符串替换为所有指定的字符串。另外，替换运算可以实现删除运算，方法也很简单，即将参数字符串设置为空。例如下面的代码。

```
<%@ page language="java" import="java.util.regex.*" contentType="text/html;
charset=gb2312"%>
<!DOCTYPE HTML PUBLIC "-//w3c//dtd html 4.0 transitional//en">
<html>
<head>
<title>8 JSP</title>
</head>
<body bgcolor="#FFFFFF">

<%
String regEx="a+";                //表示一个或多个 a

Pattern p=Pattern.compile(regEx);

Matcher m=p.matcher("aaabbced a ccdeaa");

String s=m.replaceAll("A");
out.print(s);
out.print("<br>");
s=m.replaceAll("");                //删除所有字符 a
out.print(s);

%>

</body>
</html>
```

正则表达式 **a+** 表示一个或多个字母 **a**。使用 **replaceAll** 方法，第一次将所有匹配的结果替换为 **"A"**，第二次则将所有匹配的结果替换为空，从而达到删除所有字母 **a** 的目的。程序运行结果如图 14.8 所示。

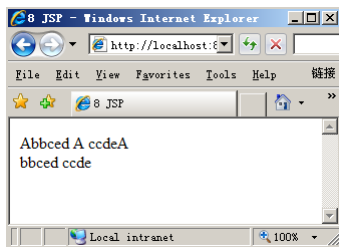


图 14.8 程序运行结果

14.2 JSP 中正则表达式的常见应用示例

本节将给出几个常见的 JSP 中正则表达式应用的示例，认真学习并动手运行这些示例，可以进一步学习正则表达式在 JSP 中的应用。

14.2.1 电子邮件地址的校验

目前，许多网站是需要用户使用电子邮件地址进行注册的，因此电子邮件地址的校验就显得非常重要。当用户输入的电子邮件地址不正确时，若系统能及时给出反馈信息，则可使网站更加人性化。这时正则表达式就可以上场了。以下示例演示了如何对电子邮箱地址进行校验。该示例包含 9.htm 和 9.jsp 两个文件。9.htm 代码如下所示。

```
<html>
<head>
<title>JSP 中的邮件地址验证</title>
</head>
<body>
<form action="9.jsp" method="POST">
输入邮件地址: <input type="text" name="email" />
<br>

<input type="submit" value="提交" />
</form>
</body>
</html>
```

9.jsp 代码如下所示。

```
<%@ page language="java" import="java.util.regex.*" contentType="text/html;
charset=gb2312"%>
<!DOCTYPE HTML PUBLIC "-//w3c//dtd html 4.0 transitional//en">
<html>
<head>
<title>9 JSP</title>
</head>
<body bgcolor="#FFFFFF">
<%
    String input = request.getParameter("email");
    //out.print(input);
    String o=input;
    // 检测输入的 EMAIL 地址是否以 非法符号 "." 或 "@" 作为起始字符
    Pattern p = Pattern.compile("^\\.|^\@"); //创建一个模式匹配器
    Matcher m = p.matcher(input); //input 为待匹配的字符串
    //out.print(m.find());
    if (m.find()) {
        out.print("EMAIL 地址不能以 '.' 或 '@' 作为起始字符");
    }else{
        // 检测是否以 "www." 为起始
        p = Pattern.compile("^www\\."); //修改模式匹配器
        m = p.matcher(input); //input 为待匹配的字符串
        if (m.find()) {
            out.print("EMAIL 地址不能以 'www.' 起始");
        }else{
            // 判断是否符合基本的邮件地址
            p = Pattern.compile("\\w+([-+.]\\w+)*@\\w+([-.]\\w+)*\\.\\w+([-.]\\w+)*");
            m = p.matcher(input); //input 为待匹配的字符串
```

```

        if (!m.find()) {
            out.print("E-mail 地址格式不正确");
            out.print("<br>");
            out.print("您现在输入的是 " + o);
        }
        else out.print("输入邮件地址符合规则");
    }
}

%>

</body>
</html>

```

在网页 9.htm 中输入电子邮件地址提交到 9.jsp 中进行验证, 在 9.jsp 中使用了三种方法对输入的邮件地址进行验证。程序运行结果如图 14.9、图 14.10、图 14.11 和图 14.12 所示。



图 14.9 输入邮件地址一



图 14.10 验证结果一



图 14.11 输入邮件地址二

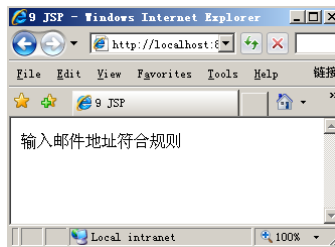


图 14.12 验证结果二

14.2.2 URL 地址的校验

同样, 正则表达式可以验证 URL 地址是否合法, 以下实例实现了对 URL 的简单校验。该实例包括 10.htm 和 10.jsp 两个文件。其中, 10.htm 的代码如下所示。

```

<html>
<head>
<title>JSP 中的 URL 地址验证</title>
</head>
<body>
<form action="10.jsp" method="POST">
输入 URL 地址: <input type="text" name="url1A" />
<br>

<input type="submit" value="提交" />
</form>
</body>
</html>

```

10.jsp 的代码如下所示。

```

<%@ page language="java" import="java.util.regex.*" contentType="text/html;

```

```

charset=gb2312"%>
<!DOCTYPE HTML PUBLIC "-//w3c//dtd html 4.0 transitional//en">
<html>
<head>
<title>10 JSP</title>
</head>
<body bgcolor="#FFFFFF">
<%
    String input = request.getParameter("urlA");

    Pattern p = Pattern.compile("[a-zA-z]+://[^\s]*"); //创建一个模式匹配器
    Matcher m = p.matcher(input); //input 为待匹配的字符串
    //out.print(m.find());
    if (!m.find()) {
        out.print("您输入的 URL 地址不符合规则");
        out.print("<br>");
        out.print("你现在输入的是: "+input);

    }else{
        out.print("您输入的 URL 正确符合规则");
    }

%>

</body>
</html>

```

在 10.jsp 中，使用正则表达式 `[a-zA-z]+://[^\s]*` 判断输入的字符串是否包含是以字母开头、字母至少出现一次、并紧跟着字符串“://”，且其后不包括任何空字符。

程序运行结果如图 14.13 和图 14.14 所示。



图 14.13 输入 URL 地址

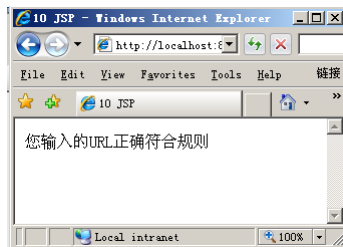


图 14.14 验证结果

14.2.3 电话号码的校验

本实例将给出完整的电话号码的验证。该实例包括 11.html 和 11.jsp 两个文件。其中，11.htm 文件代码如下所示。

```

<html>
<head>
<title>JSP 中的电话号码验证</title>
</head>
<body>
<form action="11.jsp" method="POST">
输入电话: <input type="text" name="phone" />
<br>

<input type="submit" value="提交" />

```

```
</form>
</body>
</html>
```

11.jsp 文件代码如下所示。

```
<%@ page language="java" import="java.util.regex.*" contentType="text/html;
charset=gb2312"%>
<!DOCTYPE HTML PUBLIC "-//w3c//dtd html 4.0 transitional//en">
<html>
<head>
<title>11 JSP</title>
</head>
<body bgcolor="#FFFFFF">
<%
    String inputPhone = request.getParameter("phone");

    Pattern p1 = Pattern.compile("\\d{3}-\\d{8}|\\d{4}-\\d{7}");

    Matcher m1= p1.matcher(inputPhone);           //input 为待匹配的字符串
    //out.print(m1.find());
    if (!m1.find()) {
        out.print("您输入的电话号码地址不符合规则");
        out.print("<br>");
        out.print("你现在输入的是: "+inputPhone);

    }else{
        out.print("您输入的电话号码正确符合规则");
    }

%>

</body>
</html>
```

在 11.jsp 中, 使用正则表达式 `\\d{3}-\\d{8}|\\d{4}-\\d{7}` 判断输入的字符串 `inputPhone` 是否符合电话号码规则。程序运行结果如图 14.15、图 14.16、图 14.17 和图 14.18 所示。

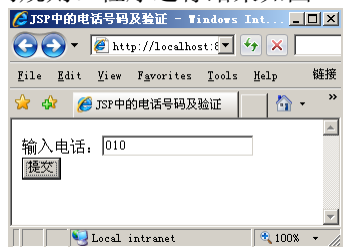


图 14.15 输入电话号码一

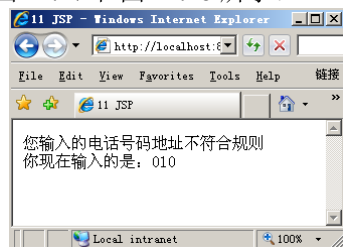


图 14.16 验证结果

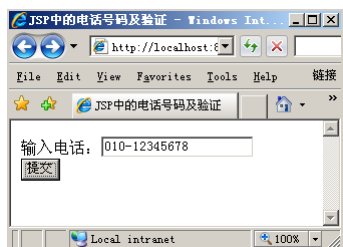


图 14.17 输入电话号码二

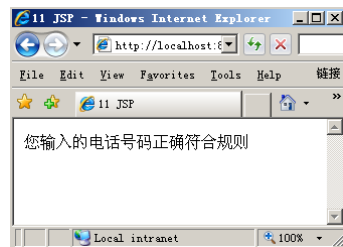


图 14.18 验证结果二

14.3 小结

本章详细讲解了正则表达式在 JSP 中的应用，如常见的邮件地址验证、电话号码验证等。在 JSP 中，使用正则表达式是比较简单的，但前提是读者要对正则表达式有深入地理解。因此，读者如果想要完成对复杂字符串的处理，则需要认真学习正则表达式知识，并多动手编写运行程序，才能更进一步地理解正则表达式。

第 5 篇 JavaScript 正则表达式应用

第 15 章 常见的 JavaScript 中数据类型及其转化

JavaScript 客户端脚本语言的应用是最广泛的，所有的浏览器都可以很好地支持 JavaScript。所以，大部分客户端的脚本都采用该语言。JavaScript 是一种弱类型的脚本语言，不仅数据类型简单，而且转化也非常方便。本章将简单地介绍其数据类型，以及常见的数据类型转化的内容。

15.1 常见的三种 JavaScript 数据类型

JavaScript 使用三种基本类型数据——数字、字符串和布尔值。此外，它还支持两种小数据类型 `null` 和 `undefined`，它们各自只定义了一个值，分别为“空”和“未定义”。除上述基本类型外，JavaScript 还支持更复杂的复合数据类型——对象。其中，与正则表达式相关的主要有三种基本类型。本将详细讲解这三种类型。

15.1.1 数字基本类型

数字是 JavaScript 最基本的类型。JavaScript 不区分整型数据和浮点型数据。它所有的数字都是以浮点数表示的，数值范围为 $-5 \times 10^{-324} \sim 19.7976931348623157 \times 10^{303}$ 。JavaScript 支持的数字类型主要包括整型、浮点型等多种数字直接量，下面将分别进行介绍。

1. 整型数

JavaScript 可以精确地识别 $-2^{53} \sim 2^{53}$ 之间的所有整数。超过这一范围的整数就会失去尾数的精确性。由于 JavaScript 中的某些运算（如位运算）只能对 32 位整数执行，所以其范围缩小为： $-2^{31} \sim 2^{31}-1$ 。以下都是合法的整型数：

```
0, -3, 1E6, 0X12F, 0111
```

这些数字都在 JavaScript 整型数允许的范围内，都可以被 JavaScript 识别。这些数字的表示形式并不一致。下面依次对几种类型进行说明。

- “0” 为普通格式。
- “-3” 是负数。
- “1E6” 也可以写作 “1e6”，是科学计数法，表示 10^6 。这种计数法要求小数点的左面为整数或小数，右边必须为整数（正负均可），如 19.23E10，表示 19.23×10^{10} 。
- “0X12A” 为十六进制表示法，类似于 C++ 等语言。JavaScript 中，在 0X 后添加合法的十六进制数字就可以表示十六进制数。
- “0111” 是八进制数。与十六进制表示法类似，在 0 之后添加合理的八进制数字就可以表示八进制数，可以为 JavaScript 所识别。

注意：十六进制数和八进制数可以为负，但不能带有小数位，同时也不能以科学计数法表示。ECMAScript 并不支持八进制数的直接量，但某些 JavaScript 的实例可以支持，因此使用时要小心。因为预先并不知道 JavaScript 会将其解释为八进制数还是十进制数。

2. 浮点型

浮点数通常表示为整数部分+小数点+小数部分。它也可以采用科学计数法来表示。科学计数法就是在实数后紧跟字母 e 或 E，字母后跟一个整型指数，这就可以表示该数值是前面的实数乘以 10 的指数次幂。

3. 特殊的数值

算术运算过程中可能会因为某些不确定因素导致错误结果的产生。例如，得到了一个超出 JavaScript 所能表示的数值范围的数据，或是某些特殊运算（如 0/0）产生了未定义的结果时，但程序仍然要求一个返回值。为了解决这一问题，JavaScript 定义了一类特殊的数值常量，用它们来表示那些常规方式无法表示的数据。表 15.1 列出了这些特殊数值常量。

表 15.1 JavaScript 中特殊数值常量

常量	含义
Infinity	表示无穷大的特殊值
NaN	特殊非字符值
Number.MAX_VALUE	可表示的最大数字
Number.MIN_VALUE	可表示的最小数字（接近于零）
Number.NaN	特殊的非数字值
Number.POSITIVE_INFINITY	表示正无穷大的特殊值
Number.NEGATIVE_INFINITY	表示负无穷大的特殊值

15.1.2 字符串基本类型

JavaScript 中的字符串（string）一般是由 Unicode 字符、数字和标点符号等组成的序列。它用来表示文本的数据类型。JavaScript 是用单引号或双引号包含来表示字符串常量的。程序代码中的字符串都是包含在单引号或双引号中的。需要注意的是，JavaScript 是没有单独的字符常量的，在 JavaScript 中，要表示单个字符时，必须采用长度为 1 的字符串。

1. 字符串直接量

JavaScript 中的字符串是由单引号或双引号括起来的 Unicode 字符序列。由单引号包含的字符串中可以含有双引号，而双引号包含的字符串中也可以含有单引号。通常，一个字符串是写在一行当中的。如果字符串太长，为了书写美观而在字符串中间添加了换行符，虽然可以将字符串放在两行当中，但程序运行时就会出现问题。因为 JavaScript 解释器会将一行末尾的换行符作为语句结束符来处理，从而使字符串的值发生改变。

由于 HTML 语言经常和 JavaScript 语言混合使用，并且在 HTML 语言中字符串也是以单引号和双引号标识的，使得程序易读性变差。为了让程序更容易阅读，最好在同时使用两种语言时，对 HTML 中的字符串采取一种引用方式，对 JavaScript 中的字符串采取另一种引用方式。

2. 转义字符

转义字符用来代表一些特殊的字符。例如，某些不能打印的字符，如回车、换行等。还有一些在 JavaScript 中具有特殊意义的符号，为了避免歧义，必须采用特殊的表达方式才能输入给浏览器，如“\”。

JavaScript 中的反斜线“\”具有特殊的用途。在反斜线的后面添加一个字符就可以用来表示一些特殊的字符。如“\n”表示换行符。另一个常用的转义字符是“\'”，它表示一个单引号。由于字符串是以单引号或双引号来标识的，当字符串中必须出现单引号时，就要采用转义字符的方式。代码如下：

```
"Watch out! \n It\'s dangerous there."
```

以上代码中的“\'”是英文缩写的标记，虽然出现在字符串中，但不再标识字符串的结束位置。表 15.2 中给出了 JavaScript 中的常用的转义字符。

表 15.2 JavaScript中的转义字符

常用序列	Unicode序列	代表字符
\0	\u0000	NUL 字符
\b	\u0008	退格符
\t	\u0009	水平制表符
\n	\u000A	换行符
\v	\u000B	垂直制表符
\f	\u000C	换页符
\r	\u000D	回车符
\'	\u0022	双引号
\'	\u0027	单引号或撇号
\\	\u005C	反斜线
\xXX		由两位十六进制数XX指定的Latin-1 字符
\uXXXX		由四位十六进制数XXXX指定的Unicode 字符

15.1.3 布尔值基本类型

在 JavaScript 中，布尔值的判断结果通常是有和无、真和假。它有且只有两个取值 True 和 False。当结果为“真”时，取值为 True；结果为“假”时，取 False。布尔值通常用来作为 JavaScript 语句结构的控制条件。

15.2 数据类型转化

JavaScript 支持弱类型变量，其变量类型并不固定。JavaScript 允许用户对不同类型的数据进行一些特殊的运算。通过这些运算可以方便地实现数据类型的转换。

15.2.1 基本数据类型转换

JavaScript 的变量没有预定类型，这使得变量能够作为数据类型参与运算，不必担心程序出现异常。对于数值、字符串和布尔值等基本数据类型，JavaScript 采取表 15.3 中的转换规则进行类型转换。

表 15.3 JavaScript中的基本数据类型转换规则

运算	结果
数值与字符串相加	将数值强制转换为字符串
布尔值与字符串相加	将布尔值强制转换为字符串
数值与布尔值相加	将布尔值强制转换为数值

要显式地将字符串转换为整数，可以使用 `parseInt` 方法；要显式地将字符串转换为数字，可以使用 `parseFloat` 方法。JavaScript 在比较大小时将字符串自动转换为相等的数字，但使用加法（连接）运算时保留为字符串。

15.2.2 将字符串转化为整数

在 JavaScript 中，使用 `parseInt` 可以将字符串转化为整数。其语法如下。

```
parseInt(numString, [radix])
```

其中，`numString` 是必选项，表示要转换为数字的字符串。`radix` 是可选项。范围在 2~36 之间，表示 `numString` 所保存数字的进制的值。如果没有提供，则前缀为 `0x` 的字符串被当作十六进制，前缀为 `0` 的字符串被当作八进制，所有其他字符串都被当作是十进制的。`parseInt` 只是返回字符串中从首位字符算起所有出现的连续数字组成的整数。如果字符串不是以数字开头，那么将返回 `NaN`。代码如下。

```
var a="123asds456adf",b="abcd123";  
document.write(parseInt(a));           //页面上打印 123  
document.write(parseInt(b));           //页面上打印 NaN
```

15.2.3 将字符串转化为浮点数

在 JavaScript 中，可以使用 `parseFloat` 方法将字符串转化为浮点数。其语法如下。

```
parseFloat(numString)
```

其中，`numString` 表示要转换为浮点数的字符串。`parseFloat` 只是返回字符串中从首位字符算起的所有出现的连续数字组成的浮点数。如果字符串不是以数字开头，那么将返回 `NaN`。

第 16 章 常见 JavaScript 字符串和数组处理

程序员在编写代码时，不可避免地要处理一些字符串数据，在 JavaScript 中，内置对象 String 专门用于字符串处理，而数组也是一种字符集合，需要使用 Array 对象进行数组运算。本章将详细介绍 String 对象和 Array 对象的常用方法和属性。

16.1 6 种字符串格式处理

字符串格式处理主要是对字符串的各种格式进行简单处理。通过这些处理，可以实现简单的数据检验功能，并可以为字符串进行预处理，从而方便后面的正则表达式匹配。

16.1.1 获取字符串的长度

在 JavaScript 中，使用 length 属性获取字符串的长度，即字符串中字符的个数。其语法如下。
string.length

示例如下。

```
<script>
var str="English 中文"+String.fromCharCode(8);
document.write(str+"<br>"+"长度为"+str.length);
</script>
```

运行这段代码，效果如图 16.1 所示。

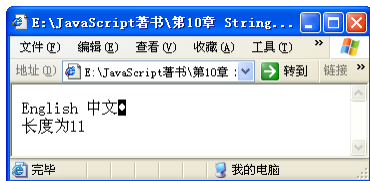


图 16.1 获取字符串的长度

16.1.2 根据指定的 Unicode 编码返回一个字符串

在 JavaScript 中，fromCharCode 方法可以根据指定的一个或多个 Unicode 编码，返回这些编码所对应的字符所组成的字符串。在获取一些特殊字符时使用该方法特别方便。其语法如下。

```
String.fromCharCode(unicode1[,unicode2[,...[,ucoden]]])
```

其中，unicode1、unicode2、...、ucoden 为指定的 Unicode 编码。fromCharCode 方法为 String 对象的静态函数，它的调用格式固定为 String.fromCharCode，不可以通过 String 对象的实例或字符串变量或常量来调用该方法。下面的代码利用 fromCharCode 方法输出了 256 个 Unicode 编码所对应的字符。

```
<script>
var i=0;
for(;i<=255;i++)
```

```

{
    document.write(String.fromCharCode(i));
}
</script>

```

运行代码，效果如图 16.2 所示。

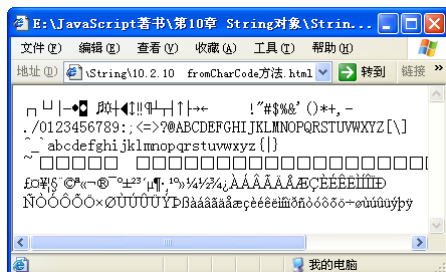


图 16.2 fromCharCode 方法演示

16.1.3 将字符串分割并存储到数组中

在 JavaScript 中，split 方法可以根据指定的分隔符，将一个字符串分割为若干部分存储到一个字符串数组中。然后，该数组将被作为该方法的返回值返回。其语法如下。

```
objString.split([separator[,limit]])
```

其中，objString 为必选项，表示字符串变量或常量。separator 为可选项，用于划分字符串的分隔符。limit 为可选项，用于显示限制返回的数组中的项数。

指定的分隔符将被作为划分字符串的标志，它们将不被保存到返回的数组中。如果没有指定分隔符或者指定的分隔符没有在字符串中找到，则整个字符串将被作为数组的一个元素，而且这个数组只有这一个元素。下面的代码利用 split 方法实现了字符串的倒序输出。

```

<script>
    var str="how are you"
    var strarr=str.split(" ")
    document.write(str);
    document.write("<br>");
    //按单词反序输出字符串
    for(var i=strarr.length-1;i>=0;i--)
    {
        document.write(strarr[i]);
        document.write(" ");
    }
</script>

```

运行代码，效果如图 16.3 所示。

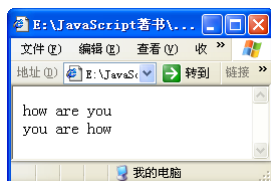


图 16.3 将字符串分割并存储到数组中

16.1.4 比较两个字符串的大小

在 JavaScript 中，localeCompare 方法可以按当前计算机的区域设置对两个字符串进行比较。

比较的结果以整数的形式返回，其可能的值有三个：-1、0 和 1。

```
string.localeCompare(string2)
```

其中，string1 为必选项，表示字符串常量或者变量。string2 为必选项，表示字符串常量或者变量。如果 string1 排在 string2 的前面，则该方法的返回值为-1；如果 string1 与 string2 相等，则该方法的返回值为 0；否则，该函数的返回值为 1。以下代码演示了 localeCompare 方法的使用。

```
<script>
    var str1="bc"
    var str2="ab"
    var str3="cd"
    strCompare(str1,str2);
    document.write("<br>");
    strCompare(str1,str3);
    function strCompare(str1,str2)
    {
        var cmp=str1.localeCompare(str2);
        var strcmp;
        if(cmp==0)
            strcmp="等于";
        else if (cmp>0)
            strcmp="大于";
        else if (cmp<0)
            strcmp="小于";
        document.write(str1+strcmp+str2);
    }
</script>
```

运行代码，效果如图 16.4 所示。

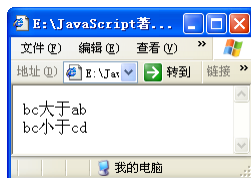


图 16.4 比较两个字符串的大小

16.1.5 将字符串转化为小写格式

在 JavaScript 中，toLowerCase 方法的作用是将字符串中的所有大写字母全部转化为小写字母。因此，调用 toLowerCase 方法处理之后，字符串中的所有字符全部为小写格式。其语法如下。

```
objString.toLowerCase()
```

以下代码演示了 toLowerCase 方法的使用。

```
<script>
    var str="aBCDef3eRTsd"
    document.write(str);
    document.write("<br>");
    document.write(str.toLowerCase());
</script>
```

运行代码，效果如图 16.5 所示。

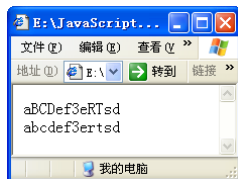


图 16.5 将字符串转化为小写格式

16.1.6 将字符串转化为大写格式

在 JavaScript 中，`toUpperCase` 方法的作用与 `toLowerCase` 方法的作用相反。该方法是将字符串中的全部小写字母转化为大写字母。其语法如下。

```
objString.toUpperCase()
```

以下代码演示了 `toUpperCase` 方法的使用。

```
<script>
var str="aBCDef3eRTsd"
document.write(str);
document.write("<br>");
document.write("toLowerCase: "+str.toLowerCase());
document.write("<br>");
document.write("toUpperCase: "+str.toUpperCase());
</script>
```

运行代码，效果如图 16.6 所示。



图 16.6 将字符串转化为大写格式

16.2 最基本的字符串查找、替换

字符串查找是指在字符串中查找特定的字符。它是最基本的字符串匹配。合理使用这些函数，可以高效完成各种匹配工作。本节主要讲解两种字符串查找方法和一种字符串替换方法。

16.2.1 获取指定字符（串）第一次在字符串中出现的位置

在 JavaScript 中，`indexOf` 方法的返回值为某一整数，代表了指定的字符（或者字符串）在字符串中第一次出现时的位置。如果在字符串中没有找到指定的字符（或字符串），则该方法的返回值为 -1。其语法如下。

```
string.indexOf(substring[,startpos])
```

其中，`string` 为必选项，表示字符串常量或变量，在其中查找子字符串。`substring` 为必选项，表示要查找的字符或字符串。`startpos` 为可选项，表示在字符串中查找时的起始位置。如果指定的值为负数，则将被视为 0；如果超出了可能的最大索引位置，则被视为该最大索引位置。以下代码演示了 `indexOf` 方法的使用。

```
<script>
var strMsg="你好，我的 QQ 是 10000，电话是 88888888，常联系哦"
var isNumber=false;
var intNum=0;
document.write(strMsg);
document.write("<br>indexOf 方法演示....");
//循环判断字符串中字符是否为数字
for(var i=0 ;i<strMsg.length;i++)
{
    var char=strMsg.charAt(i);
```

```

    if(isNumeric(char))
    {
        if(! isNumber)
        {
            isNumber=true;
            intNum++;
            document.write("<br>");
        }
        document.write(char);
    }
    else
        isNumber=false;
}
if(!isNumber)
    document.write("<br>");
//输出共找到了几个数字串
document.write("共找到"+intNum+"个数字串");
//自定义函数, 判断是否为数字
function isNumeric(ch)
{
    var numbers="0123456789";
    if(numbers.indexOf(ch)!=-1)
        return true;
    else
        return false;
}
</script>

```

运行代码, 效果如图 16.7 所示。

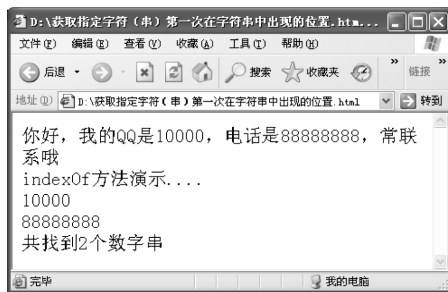


图 16.7 获取指定字符（串）第一次在字符串中出现的位置

16.2.2 获取指定字符（串）最后一次在字符串中出现的位置

在 JavaScript 中, `lastIndexOf` 方法的返回值为某一整数, 代表了指定的字符（或者字符串）在字符串中最后一次出现时的位置。如果在字符串中没有找到指定的字符（或字符串）, 则该方法的返回值为 -1。其语法如下。

```
string.lastIndexOf(substring[, startpos])
```

其中, `string` 为必选项, 表示字符串常量或变量, 在其中查找子字符串。`substring` 为必选项, 表示要查找的字符或字符串。`startpos` 为可选项, 表示在字符串中查找时的起始位置。如果指定的值为负数, 则将被视为 0; 如果超出了可能的最大索引位置, 则被视为该最大索引位置; 如果没有指定该参数, 则默认为自后向前查找。

16.2.3 替换字符串中指定的内容

在 JavaScript 中，`replace` 方法可以将字符串中一些子串用指定的内容替换掉。替换后的新的字符串将被作为 `replace` 方法的返回值返回。其语法如下。

```
string.replace(bereplaced,toreplace)
```

其中，`string` 为必选项，表示字符串常量或变量。`bereplaced` 为必选项，表示字符串中将被替换掉的内容。`toreplace` 为必选项，表示来替换指定字符串内容的新串。下面的代码演示了如何使用 `replace` 方法替换字符串中的内容。

```
<form>
  <textarea name="txt" cols="28" rows="5"></textarea>
  <br>
  字符串<input type='text' name='bere'><br>
  替换为<input type='text' name='tore'>
  <input type="button" value="替换" onClick="retxt()">
</form>
<script>
  function retxt()
  {
    var bere=document.forms[0].bere.value;
    var tore=document.forms[0].tore.value;
    var txt=document.forms[0].txt.innerText;
    var retxt=txt.replace(bere,tore);
    //循环替换，直到全部替换完成
    do
    {
      txt=retxt;
      retxt=retxt.replace(bere,tore);
    }
    while(txt!=retxt)
    document.forms[0].txt.innerText=txt;
  }
</script>
```

因为 `replace` 方法只替换第一次匹配的字符串，所以本例中使用了 `do-while` 循环，以保证查找并替换所有的匹配情况。运行这段代码并在文本和文本域中填写相应内容，如图 16.8 所示，单击“替换”按钮后，可以看到如图 16.9 所示的页面。



图 16.8 替换前页面

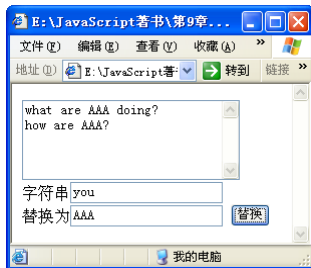


图 16.9 替换后页面

16.3 字符串截取、组合的方法

字符串截取是指获取字符串的部分。组合是将多个字符串组合成一个新的字符串。本节将详细介绍如何截取、组合字符串。

16.3.1 返回字符串中指定位置处的字符

在 JavaScript 中, `charAt` 方法可以获取字符串中指定位置的字符。字符串中字符位置从 0 开始计算, 最后一个位置为字符串的长度减 1。如果指定的位置超过了这个范围, 则返回空字符。其语法如下。

```
objString.charAt(index)
```

其中, `objString` 为必选项, 表示 `String` 对象实例的名称。`index` 为必选项, 表示字符索引, 也就是字符位置, 第一个字符的位置为 0。下面的代码演示了如何用 `charAt` 方法从字符串中提取数字。

```
<script>
var strMsg="你好, 我的 QQ 是 1000, 电话是 8888888, 常联系哦"
var isNumber=false;
var intNum=0;
document.write(strMsg);
document.write("<br>分析中....");
//循环判断字符串中字符是否为数字
for(var i=0 ;i<strMsg.length;i++)
{
    var char=strMsg.charAt(i);
    //判断字符 char 是否为数字
    if(parseInt(char)>=0)
    {
        if(! isNumber)
        {
            isNumber=true;
            intNum++;
            document.write("<br>");
        }
        document.write(char);
    }
    else
        isNumber=false;
}
if(!isNumber)
    document.write("<br>");
//输出共找到了几个数字串
document.write("共找到"+intNum+"个数字串");
</script>
```

上述代码利用 `charAt` 方法和 `parseInt` 函数, 实现了从字符串中识别数字串的功能, 同时还可以统计所找到的数字串的个数, 每找到一个数字串就单独作为一行输出。运行代码, 效果如图 16.10 所示。

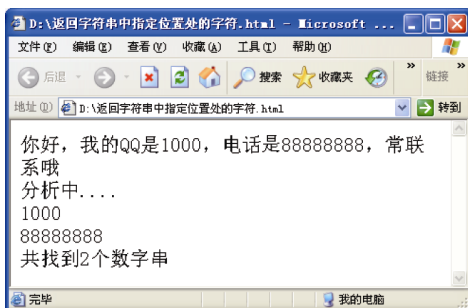


图 16.10 返回字符串中指定位置处的字符

16.3.2 将一个或多个字符串连接到当前字符串的末尾

在 JavaScript 中，concat 方法可以将一个或多个字符串添加到当前字符串的末尾。如果指定的参数不是字符串，则它将首先被转化为字符串。其语法如下。

```
string1.concat([string2[,string3[,...[,stringN]]]])
```

其中，string1 为必选项，表示字符串变量或常量。string2、...、stringN 为可选项，表示将被添加到 string1 末尾的内容。下面的代码演示了 concat 方法的使用。

```
<script>
    var str1="hello";
    str1=str1.concat(" this is a test ", "<b>", "haha ", 123, "</b>");
    document.write(str1);
</script>
```

运行代码，效果如图 16.11 所示。

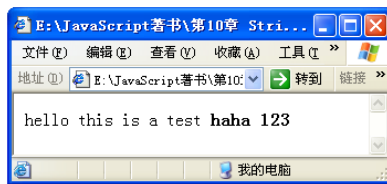


图 16.11 将一个或多个字符串连接到当前字符串的末尾

从执行结果可以看出，和没有显示出来，这是因为它们被解释成了 HTML 标记。所以，这两个字符串之间的字符明显变粗了。

16.3.3 获取指定位置的字符的 Unicode 编码

在 JavaScript 中，charCodeAt 方法与 charAt 方法不同，其返回值不是字符串中指定位置处的字符，而是该字符的 Unicode 编码。其语法如下。

```
objString.charCodeAt(index)
```

其中，objString 为必选项，表示字符串变量或常量。index 为必选项，表示字符索引，即字符在字符串中的位置。第一个字符的位置为 0，第二个位置为 1，依此类推。如果指定的位置超出了该范围，则该方法的返回值为空。下面的代码演示了 charCodeAt 方法的使用。

```
<script>
    var strMsg="你好，我的 QQ 是 10000，电话是 88888888，常联系哦"
    var isNumber=false;
    var intNum=0;
    document.write(strMsg);
    document.write("<br>charCodeAt 方法演示....");
    //循环判断字符串中字符是否为数字
    for(var i=0 ;i<strMsg.length;i++)
    {
        var char=strMsg.charAt(i);
        if(isNumeric(char.charCodeAt(0)))
        {
            if(! isNumber)
            {
                isNumber=true;
                intNum++;
                document.write("<br>");
            }
            document.write(char);
        }
    }
}
```

```

    }
    else
        isNumber=false;
}
if(!isNumber)
    document.write("<br>");
//输出共找到了几个数字串
document.write("共找到"+intNum+"个数字串");
//自定义函数,判断是否为数字
function isNumeric(ch)
{
    if(ch>=48&&ch<=57)
        return true;
    else
        return false;
}
</script>

```

上述代码通过 Unicode 编码的返回来判断一个字符是否为数字。运行代码,效果如图 16.12 所示。

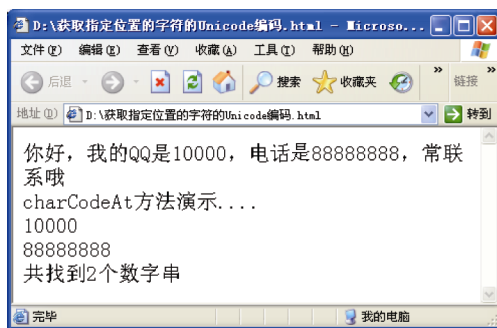


图 16.12 获取指定位置的字符的 Unicode 编码

16.3.4 从字符串中提取子串 (1)

在 JavaScript 中, substring 方法可以根据指定的起始位置和结束位置,从字符串中提取一个子串。子串的内容和长度受指定的参数影响。其语法如下。

```
string.substring(start[,end])
```

其中, string 为必选项,表示要从中提取子串的字符串。start 为必选项,类型为整数,指定提取子串的起始位置。end 为可选项,类型为整数,指定提取子串的结束位置。下面的代码演示了 substring 方法的使用。

```

<script>
var str="0123456789";
with(document)
{
    write(str);
    //对比一
    write("<br>");
    write("slice(3)="+str.slice(3)+"<br>");
    write("substring(3)="+str.substring(3));
    //对比二
    write("<br>");
    write("slice(3,-3)="+str.slice(3,-3)+"<br>");
    write("substring(3,-3)="+str.substring(3,-3));
}

```

```
//对比三
write("<br>");
write("slice(3,0)="+str.slice(3,0)+"<br>");
write("substring(3,0)="+str.substring(3,0));
}
</script>
```

运行代码，效果如图 16.13 所示。

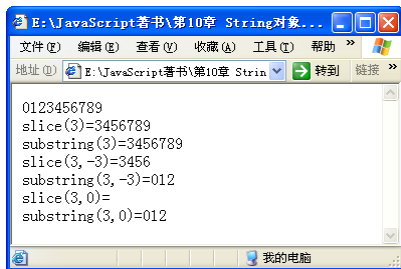


图 16.13 从字符串中提取子串（1）

16.3.5 从字符串中提取子串（2）

在 JavaScript 中，slice 方法可以根据指定的起始位置和结束位置，从字符串中提取一个子串，子串的内容和长度受指定的参数的影响。其语法如下。

```
string.slice(start[,end])
```

其中，string 为必选项，表示要从中提取子串的字符串。start 为必选项，类型为整数，指定提取子串的起始位置。end 为可选项，为整数，指定提取子串的结束位置。

如果指定的 start 的值为负数，则从 length+start 的位置开始提取（length 为字符串的长度）；如果 start 的值超出了字符串的长度，则该方法的返回值为空；如果 end 的值为负数，则结束位置为 length+end；如果 end 小于 start，则该方法的返回值为空；如果没有指定 end，则返回一个从 start 开始一直到字符串末尾的子串。下面的代码演示了 slice 方法的使用。

```
<head>
  slice 方法演示
</head>
<script>
  var str="hello,this is a test";
  var str1=str.slice(6,12);
  var str2=str.slice(6);
  var str3=str.slice(6,-6);
  with(document)
  {
    write(str1);
    write("<br>");
    write(str2);
    write("<br>");
    write(str3);
  }
</script>
```

运行代码，效果如图 16.14 所示。

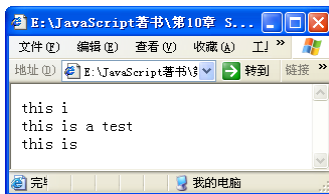


图 16.14 从字符串中提取子串 (2)

16.3.6 从字符串中提取子串 (3)

在 JavaScript 中, `substr` 方法可以根据指定起始位置和子串长度来提取子串。其语法如下。

```
string.substr(start[,length])
```

其中, `string` 为必选项, 表示字符串常量或变量, 用于从中提取子串。`start` 为必选项, 类型为整数, 用于指定提取子串时的起始位置。`length` 为可选项, 表示所要提取的子串的长度。

如果没有指定 `length` 参数, 则该方法返回从 `start` 位置开始一直到字符串结尾的子串。如果指定的参数 `length` 为 0 或者负值, 则该方法的返回值为空。下面的代码演示了 `substr` 方法的使用。

```
<script>
var str="0123456789"
with(document)
{
    write("对比一<br>");
    write(str.slice(6));
    write("<br>");
    write(str.substr(6));
    write("<br>");
    write("对比二<br>");
    write(str.slice(3,6));
    write("<br>");
    write(str.substr(3,6));
    write("<br>");
    write(str.substr(3,6-3));
}
</script>
```

运行代码, 效果如图 16.15 所示。

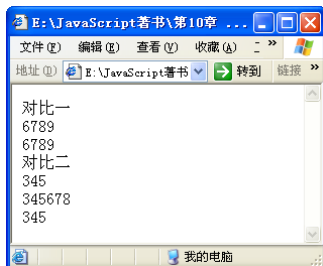


图 16.15 从字符串中提取子串 (3)

16.4 字符串 HTML 格式化

字符串 HTML 格式化可以为字符串添加各种标签。格式化后, 字符串可以更好地在网页中显示。本节将详细介绍 12 种常见的字符串 HTML 格式化方式。

16.4.1 在字符串两端加入锚点标志

在 JavaScript 中，锚点可以实现同一页面内的超级链接。`anchor` 方法可以为指定的字符串的两端添加上锚点标志，使之成为网页中的一个锚点。其语法如下。

```
objString.anchor(anchorname)
```

其中，`objString` 为必选项，表示字符串常量或变量。`anchorname` 为可选项，表示锚点的名称。

下面的代码演示了 `anchor` 方法的使用。

```
<script>
    window.alert("anchor 方法测试".anchor("test"));
</script>
```

运行代码，效果如图 16.16 所示。



图 16.16 在字符串两端加入锚点标志

16.4.2 在字符串的两端加上粗体标志

在 JavaScript 中，`bold` 方法的作用是在字符串的两端加上粗体标志，即分别在字符串的两端加上 `` 和 `` 标志。当使用 `write` 方法或 `writeln` 方法输出调用了 `bold` 方法之后的字符串时，字符串将会以粗体格式显示出来。其语法如下。

```
objString.bold()
```

下面的代码演示了 `bold` 方法的使用。

```
<script>
    objstr="bold 方法测试"
    document.write(objstr.bold());
    document.write("<br>");
    objstr=objstr.bold();
    document.write("<input type='text' value='"+objstr+"'>");
</script>
```

这段代码在页面和文本框中显示了 `bold` 方法的作用效果和返回值。运行代码，效果如图 16.17 所示。



图 16.17 在字符串的两端加上粗体标志

16.4.3 在字符串两端加入斜体标签

在 JavaScript 中，输出字符串时，调用 `italics` 方法可以获得斜体效果。因为该方法会把字符串的前后两端分别加上 `<i>` 和 `</i>` 标记。其语法如下。

```
objString.italics()
```

下面的代码演示了 `italics` 方法的使用。

```

<script>
  var str="this is a test"
  with (document)
  {
    write("原始字符串: "+str);
    write("<br>");
    write("italics 方法作用后:"+str.italics());
    write("<br>");
    write("italics 方法的返回值"+"<input type='text' size=30 value='"+str.italics()+"'>");
  }
</script>

```

上述代码分别输出了 `italics` 方法作用前和作用后的字符串输出效果, 同时输出了 `italics` 方法的返回值, 如图 16.18 所示。通过对比, 可以清楚地看到 `italics` 方法的作用效果。

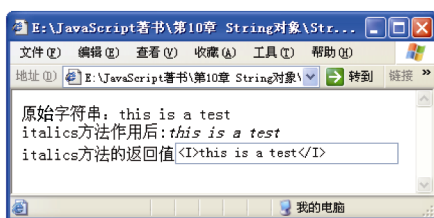


图 16.18 在字符串两端加入斜体标签

16.4.4 在指定字符串的两端加上大字体标志

在 JavaScript 中, `big` 方法可以将字符串的两端加上大字体标志, 即分别在字符串的两端加上 `<big>` 和 `</big>` 标志。这样在将字符串输出到网页中时, 该字符串就会以粗体显示。其语法如下。

```
objString.big()
```

下面的代码演示了 `big` 方法的使用。

```

<script>
  str1="big 方法测试";
  str2="big 方法测试";
  document.write(str1);
  document.write("<br>");
  document.write(str2.big());
</script>

```

上述代码分别输出了调用 `big` 方法前和调用 `big` 方法后的同一字符串, 如图 16.19 所示。通过比较可以看出 `big` 方法的作用。

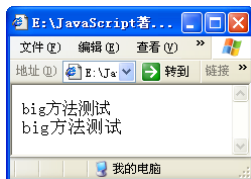


图 16.19 在指定字符串的两端加上大字体标志

16.4.5 在字符串的两端加上固定宽度字体标记

在 JavaScript 中, `fixed` 方法可以在字符串的两端加上固定宽度字体标记, 即分别在字符串的两端加上 `<tt>` 和 `</tt>` 标记。其语法如下。

```
objString.fixed()
```

下面的代码演示了 `fixed` 方法的使用。

```
<script>
  var str="this is a test"
  var str1=str.fixed();
  with (document)
  {
    write("原始字符串: "+str);
    write("<br>");
    write("fixed 方法作用后:"+str1);
    write("<br>");
    write("fixed 方法的返回值"+"<input type='text' size=25 value='"+str1+"'>");
  }
</script>
```

上述代码分别输出了 `fixed` 方法作用前和作用后的字符串输出效果，同时输出了 `fixed` 方法的返回值，如图 16.20 所示。通过对比，可以清楚地看到 `fixed` 方法的作用。

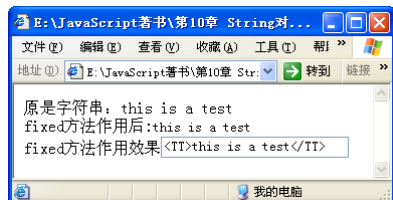


图 16.20 在字符串的两端加上固定宽度字体标记

16.4.6 设置字符串输出时的字体大小

在 JavaScript 中，`fontsize` 方法可用于设置字符串输出时的字体大小。它是通过在字符串的两端分别添加 `` 和 `` 标记来实现的。其中，`number` 为整数值，用于控制字体大小。其语法如下。

```
objString.fontsize(number)
```

其中，`objString` 为必选项，表示字符串变量或常量。`number` 为必选项，型为整数值，表示用于设置字符串字体大小。下面的代码演示了 `fontsize` 方法的使用。

```
<script>
  var str="abcdefgh"
  document.write("原始字符串: "+str);
  document.write("<br>");
  //这个输出，字体逐渐变大
  document.write("fontsize 循环设置后: ")
  for(var i=0;i<str.length;i++)
  {
    var char=str.charAt(i);
    document.write(char.fontsize(i+1));
  }
</script>
```

上述代码通过 `for` 循环，逐个获取了字符串中的字符并调用 `fontsize` 方法设置了它们的字体大小，而且字体逐渐变大。运行代码，效果如图 16.21 所示。

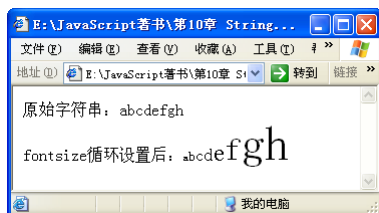


图 16.21 设置字符串输出时的字体大小

16.4.7 设置字符串输出时的前景色

在 JavaScript 中, `fontcolor` 方法可用于设置字符串输出时的颜色。它是通过在字符串的两端分别添加 `` 和 `` 标记来实现的。其中, `color` 为颜色名称或颜色值。其语法如下。

```
objString.fontcolor(color)
```

其中, `objString` 为必选项, 表示字符串变量或常量。`color` 为必选项, 表示要设置的颜色名称或十六进制的 RGB 颜色值。下面的代码演示了如何使用 `fontcolor` 设置字体颜色。

```
<script>
    var str="this is a test"
    var str1=str.fontcolor("red");
    with (document)
    {
        write("原始字符串: "+str);
        write("<br>");
        write("fontcolor 方法作用后:"+str1+str.fontcolor("BC8820"));
        write("<br>");
        write("fontcolor 方法作用效果<br>"+<input type='text' size=40 value='"+
str1+"'">");
    }
</script>
```

上述代码分别输出了 `fontcolor` 方法作用前和作用后的字符串输出效果, 同时输出 `fontcolor` 方法的返回值, 如图 16.22 所示。通过对比, 可以清楚地看到 `fontcolor` 方法的作用效果。

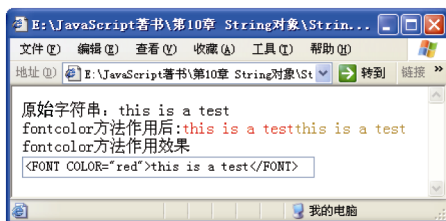


图 16.22 设置字符串输出时的前景色

16.4.8 在字符串上加入超链接

在 JavaScript 中, `link` 方法可以在指定的字符串上添加超链接。在 Web 页面上单击该链接时, 当前页面将会链接到指定页面。其语法如下。

```
string.link(url)
```

其中, `string` 为必选项, 表示字符串常量或变量。`url` 为必选项, 表示字符串表达式, 指定链接的目标 URL。下面的代码演示了 `link` 方法的使用。

```
<html>
```

```

<head>
    link 方法演示
</head>
<script>
    var str="链接测试"
    var url="http:\\\\www.upc.edu.cn"
    with (document)
    {
        write("原是字符串: "+str);
        write("<br>");
        write("link 方法作用后:"+str.link(url));
        write("<br>");
        write("link 方法的返回值 <br>"+<input type='text' size=50 value='"+
str.link(url)+"'>");
    }
</script>
</html>

```

上述代码在 URL 中之所以使用四个反斜杠（“\\”），是因为两个反斜杠（“\”）会被作为转义字符，因此输出是一个反斜杠。运行代码，效果如图 16.23 所示。

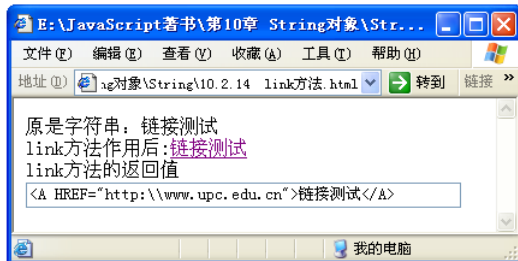


图 16.23 在字符串上加入超链接

16.4.9 在字符串两端加上小字体标记

在 JavaScript 中，small 方法的作用是在指定字符串的两端加上小字体标记，即分别在字符串的前后两侧分别加入 HTML 标记<small>和</small>。当将经过此处理的字符串输出到 Web 页面上时，它们的字体将小于普通字体。其语法如下。

```
objString.small()
```

下面的代码演示了 small 方法的使用。

```

<script>
    var str="this is a test"
    with (document)
    {
        write("原始字符串: "+str);
        write("<br>");
        write("small 方法作用后:"+str.small());
        write("<br>");
        write("small 方法的返回值 "+<input type='text' size=30 value='"+
str.small()+"'>");
    }
</script>

```

上述代码对比了通过 small 方法处理前和处理后的字符串的输出效果，同时输出 small 方法的返回值。通过对比，可以清楚地看到 small 方法的作用效果。运行代码，效果如图 16.24 所示。

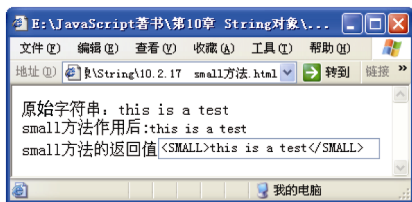


图 16.24 在字符串两端加上小字体标记

16.4.10 在字符串两端加入下标标签

在 JavaScript 中, sub 方法可以在字符串的两端分别加入 HTML 标记 _和, 将 sub 方法作用后的字符串输出到页面上之后, 该字符串将被作为下标显示出来。其语法如下。

objString.sub()

下面的代码演示了 sub 方法的使用。

```
<head>
  sub 方法演示
</head>
<script>
  var str="this is a test"
  with (document)
  {
    write("原是字符串: "+str);
    write("<br>");
    write("sub 方法作用后:"+str.sub());
    write(" A"+"1".sub()+" b"+"left".sub());
    write("<br>");
    write("sub 方法的返回值 "+<input type='text' size=30 value='"+
str.sub()+"'>");
  }
</script>
```

运行代码, 效果如图 16.25 所示。

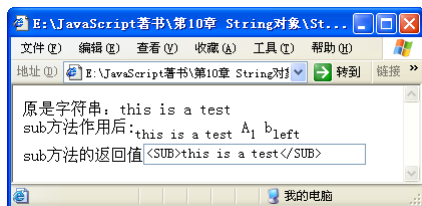


图 16.25 在字符串两端加入下标标签

16.4.11 在字符串两端加入上标标签

在 JavaScript 中, sup 方法可以在字符串的两端分别加入 HTML 标记 ^和。将 sup 方法作用后的字符串输出到页面上之后, 该字符串将被作为上标显示出来。其语法如下。

objString.sup()

下面的代码演示了 sup 方法的使用。

```
<script>
  var str="this is a test"
  with (document){
    write("原始字符串: "+str);
    write("<br>");
```

```

write("sup 方法作用后:"+str.sup());
write("  A"+"1".sup()+"  b"+"left".sup());
write("<br>");
write("sup 方法的返回值"+"<input type='text' size=30 value='"+str.sup()+
"'>");}
</script>

```

运行代码，效果如图 16.26 所示。

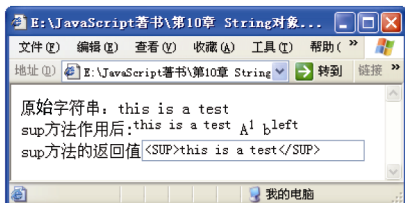


图 16.26 在字符串两端加入上标标签

16.4.12 在字符串的两端加入下划线标记

在 JavaScript 中，`strike` 可以分别在字符串的前后两端加入`<strike>`和`</strike>`标记。当使用 `write` 法或 `writeln` 方法输出时，就会在字符串的上面出现删除线。其语法如下。

```
objString.strike()
```

下面的代码演示了 `strike` 方法的使用。

```

<script>
var str="this is a test"
with (document)
{
    write("原始字符串: "+str);
    write("<br>");
    write("strike 方法作用后:"+str.strike());
    write("<br>");
    write("strike 方法的返回值 "+"<input type='text' size=30 value='"+
str.strike()+"'>");
}
</script>

```

运行代码，效果如图 16.27 所示。

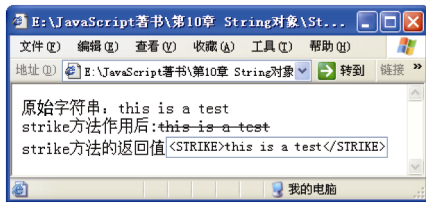


图 16.27 在字符串的两端加入删除线标记

16.5 Array 对象的方法及使用

数组是专门用来存储有序数列的。JavaScript 专门提供 `Array` 对象用来对数组进行处理。本节将详细介绍常见数组的运算。

16.5.1 连接其他数组到当前数组末尾

在 JavaScript 中, `concat` 属性可以把当前数组和指定的数组相连接, 然后返回一个新的数组。该数组中含有前面两个数组的全部元素, 其长度为两个数组的长度之和。其语法如下。

```
array.concat(array2)
```

其中, `array1` 为必选项, 表示数组名称。`array2` 为必选项, 表示数组名称, 该数组中的元素将被添加到数组 `array1` 中。下面的代码演示了 `concat` 方法的使用。

```
<script>
var array1=new Array(1,2,3,4,5,6,7);
var array2=new Array(8,9,10);
var array=array.concat(array2);
//自定义函数, 输出数组中所有数据
function writeArr(arrname,sp)
{
    for(var i=0;i<arrname.length;i++)
    {
        document.write(arrname[i]);
        document.write(sp);
    }
    document.write("<br>");
}
document.write("数组 1: ");
writeArr(array1,"");
document.write("数组 2: ");
writeArr(array2,"");
document.write("数组 3: ");
writeArr(array,"");
</script>
```

上述代码定义了两个数组 `array1` 和 `array2`, 然后把这两个数组连接并将值赋给数组 `array`。运行代码, 效果如图 16.28 所示。

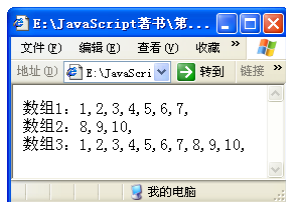


图 16.28 连接其他数组到当前数组末尾

16.5.2 将数组元素连接为字符串

在 JavaScript 中, `join` 方法将数组中所有元素连接为一个字符串。如果数组中的元素不是字符串, 则该元素将首先被转化为字符串。各个元素之间可以以指定的分隔符进行连接。其语法如下。

```
array.join(separator)
```

其中, `array` 为必选项, 表示数组的名称。`separator` 为必选项, 表示连接各个元素之间的分隔符。下面的代码演示了 `join` 方法的使用。

```
<script>
var str1="this ia a test";
var arr=str19.split(" ");
var str2=arr.join(",");
with(document){
```

```

write(str1);
write("<br>分割为数组，数组长度"+arr.length+"，重新连接如下：<br>");
write(str2);
}
</script>

```

上述代码首先使用 `split` 方法以 “ ”（空格）为分隔符将字符串分割存储到数组中，然后调用 `join` 方法以 “，”（逗号）为分隔符，将数组中的各个元素重新连接为一个新字符串。运行代码，效果如图 16.29 所示。

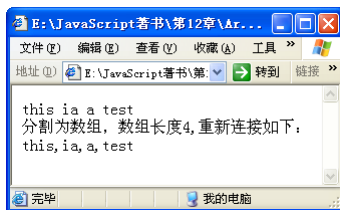


图 16.29 将数组元素连接为字符串

16.5.3 删除数组中的第一个元素

在 JavaScript 中，`shift` 方法可删除数组中的第一个元素，其返回值为该元素的值。其语法如下。

```
array.shift()
```

其中，`array` 为数组的名称，下面的代码演示了 `shift` 方法的使用。

```

<script>
var arr=new Array(1,2,3,4,5,6,7,8,9);
with (document)
{
    write(arr.join(","));
    write("<br>删除元素"+arr.shift()+"<br>");
    write("删除元素"+arr.shift()+"<br>");
    write(arr.join(","));
}
</script>

```

运行代码，效果如图 16.30 所示。

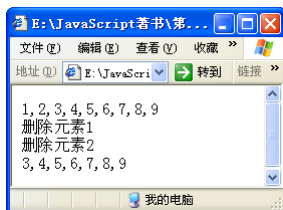


图 16.30 删除数组中的第一个元素

16.5.4 删除数组中最后一个元素

在 JavaScript 中，`pop` 方法可将数组中的最后一个元素删除并返回该元素。调用该方法后，数组的长度将减小 1。如果数组为空，则该方法的返回值为 `undefined`。其语法如下。

```
array.pop()
```

下面的代码演示了 `pop` 方法的使用。

```
<script>
```

```

var arr=new Array(1,2,3,4,5,6,7,8,9);
with (document)
{
    write(arr.join(","));
    write("<br>删除元素"+arr.pop()+"<br>");
    write("删除元素"+arr.pop()+"<br>");
    write(arr.join(","));
}
</script>

```

运行代码，效果如图 16.31 所示。

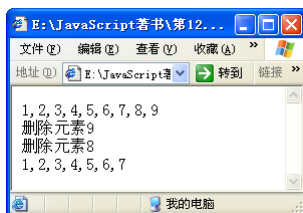


图 16.31 删除数组中最后一个元素

16.5.5 删除或替换数组中部分数据

在 JavaScript 中，splice 方法可以通过指定起始索引和数据个数的方式，删除或替换数组中的部分数据。该方法的返回值为被删除或替换掉的数据。其语法如下。

```
array.splice(start,count[,data1[,data2[,...[,datacount]]]])
```

其中，array 为必选项，表示数组名称。start 为必选项，类型为整数，表示起始索引。count 为必选项，类型为整数，表示要删除或替换的数组的个数。data 为可选项，表示用于替换指定数据的新数据。如果没有指定 data 参数，则该指定的数据将被删除；如果指定了 data 参数，则数组中的数据将被替换。下面的代码演示了 splice 方法的使用。

```

<script>
var arr=new Array(0,1,2,3,4,5,6,7,8,9,10);
var rewith=new Array("a","b","c");
var tmp1=arr.splice(2,4,rewith);
with(document)
{
    writeArr("替换了 4 个数据",tmp1);
    writeArr("替换为: ",rewith);
    writeArr("替换后",arr);
    var tmp2=arr.splice(5,2);
    writeArr("删除 2 个数据",tmp2);
    writeArr("替换后",arr);
}
//自定义函数输出提示信息和数组元素
function writeArr(str,array)
{
    document.write(str+":");
    document.write(array.join(","));
    document.write("<br>");
}
</script>

```

上述代码分别演示了如何使用 splice 方法替换和删除数组中指定数目的数据。运行代码，效果如图 16.32 所示。

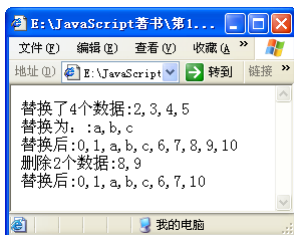


图 16.32 删除或替换数组中部分数据

16.5.6 将指定的数据添加到数组中

在 JavaScript 中，`push` 方法可以将所指定的一个或多个数据添加到数组中。该方法的返回值为添加新数据后数组的长度。其语法如下。

```
array.push([data1[,data2[,...[,datan]]]])
```

其中，`array` 为必选项，表示数组名称。`data1`、`data2`、……、`datan` 为可选参数，表示将被添加到数组中的数据。如果 `data1` 到 `datan` 中的某一参数为数组，则该数组中的所有元素将被添加到数组中。下面的代码演示了如何利用 `push` 方法向数组中添加新数据。

```
<script>
var arr=new Array();
document.write("向数组中写入数据: ");
//单个数据写入数组
for (var i=1;i<=4;i++)
{
    var data=arr.push(Math.ceil(Math.random()*10));
    document.write(data);
    document.write("个,");
}
document.write("<br>");
//批量写入数组
var data=arr.push("a",3.14,"hello");
document.write("批量写入, 数组长度已为"+data+"<br>");
var newarr=new Array(1,2,3,4,5);
document.write("向数组中写入另一个数组<br>");
//写入新数组
arr.push(newarr);
document.write("全部数据如下:<br>");
document.write(arr.join(", "));
</script>
```

上述代码使用 `push` 方法分别向数组中逐个和批量添加了数据。运行代码，效果如图 16.33 所示。

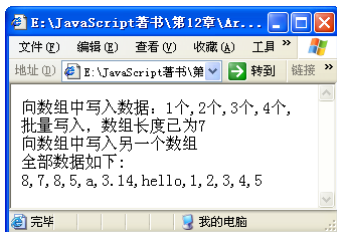


图 16.33 将指定的数据添加到数组中

16.5.7 在数组前面插入数据

在 JavaScript 中, `unshift` 方法与 `shift` 方法的作用相反。该方法是在数组的开始插入一个或多个数据, 返回值为增加了新插入数据之后的数组长度。其语法如下。

```
array.unshift([data1[,data2[,...[,datan]]]])
```

其中, `array` 为必选项, 表示数组名称。`data1` 到 `datan` 为可选项, 表示要插入到数组开始的数据。下面的代码演示了 `unshift` 方法的使用。

```
<script>
    var arr=new Array();
    //单个数据写入数组
    for (var i=1;i<=4;i++)
    {
        arr.unshift(i);
    }
    document.write("<br>");
    //批量写入数组
    arr.unshift("a",3.14,"hello");
    var newarr=new Array(1,2,3,4,5);
    //写入新数组
    arr.unshift(newarr);
    document.write("全部数据如下:<br>");
    document.write(arr.join(","));
</script>
```

运行代码, 效果如图 16.34 所示。

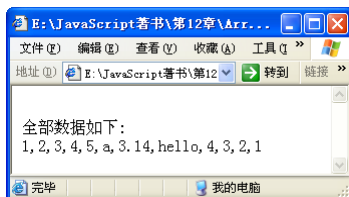


图 16.34 在数组前面插入数据

16.5.8 获取数组中的一部分数据

在 JavaScript 中, `slice` 方法可以从数组中提取一部分数据, 并将这部分数据作为一个数组返回。其语法如下。

```
array.slice(start[,end])
```

其中, `array` 为必选项, 表示数组名称。`start` 为必选项, 表示获取数据的起始索引位置, 从 0 开始。`end` 为可选项, 表示获取数据的结束位置, 从 0 开始。

该方法返回的数据中不包括 `end` 索引所对应的数据。如果 `start` (`end`) 的值为负值, 则其值将被自动替换为 `start` (`end`) + `length` (`length` 为数组长度)。如果没有指定 `end` 值, 则返回从 `start` 开始到数组末尾的所有元素。下面的代码演示了 `slice` 方法的使用。

```
<script>
    var arr=new Array(1,2,3,4,5,6,7,8,9,10);
    writeArr("原始数组 arr",arr);
    writeArr("arr.slice(2,5) 提取片断",arr.slice(2,5));
    writeArr("arr.slice(-8,-1) 提取片断",arr.slice(-8,-1));
    writeArr("arr.slice(1) 提取片断",arr.slice(1));
    //自定义函数输出提示信息和数组元素
    function writeArr(str,array)
```

```

{
    document.write(str+");
    document.write(array.join(", "));
    document.write("<br>");
}
</script>

```

运行代码，效果如图 16.35 所示。

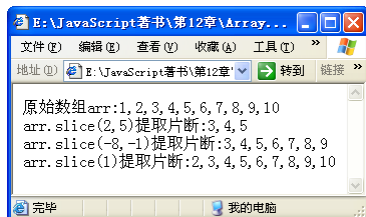


图 16.35 获取数组中的一部分数据

16.5.9 反序排列数组中的元素

在 JavaScript 中，reverse 方法可以将数组中的元素反序排列。数组中所包含的内容和数组的长度不会改变。其语法如下。

```
array.reverse()
```

其中，array 为数组的名称。下面的代码演示了 reverse 方法的使用。

```

<script>
var arr=new Array(1,2,3,4,5,6,7,8,9);
with (document)
{
    write("数组为:");
    write(arr.join(", "));
    arr.reverse();
    write("<br>反序后:");
    write(arr.join(", "));
}
</script>

```

运行代码，效果如图 16.36 所示。

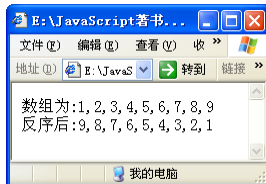


图 16.36 反序排列数组中的元素

16.5.10 对数组中的元素进行排序

在 JavaScript 中，sort 方法可以对数组中的所有元素按 Unicode 编码进行排序，并返回经过排序后的数组。sort 方法默认按升序进行排列。其语法如下。

```
array.sort([cmpfun(arg1, arg2)])
```

其中，array 为必选项，表示数组名称。cmpfun 为可选项，表示比较函数。arg1、arg2 为可选项，表示比较函数的两个参数。下面的代码演示了如何使用 sort 方法对数组中的数据进行排序。

```
<script>
```

```

var arr=new Array(2,5,3,20,1,"b","x","B","X");
writeArr("排序前",arr);
writeArr("升序排列",arr.sort());
writeArr("降序排列,字母不分大小写",arr.sort(desc));
writeArr("严格降序排列",arr.sort(desc1));
//自定义函数输出提示信息和数组元素
function writeArr(str,array)
{
    document.write(str+":");
    document.write(array.join(", "));
    document.write("<br>");
}
//按降序排列,字母不区分大小写
function desc(a,b)
{
    var a=new String(a);
    var b=new String(b);
    //如果 a 大于 b, 则返回-1, 所以 a 排在前 b 排在后
    return -1*a.localeCompare(b) ;
}
//严格降序
function desc1(a,b)
{
    var stra=new String(a);
    var strb=new String(b);
    var ai=stra.charCodeAt(0);
    var bi=strb.charCodeAt(0);
    if( ai>bi )
        return -1;
    else
        return 1;
}
</script>

```

上述代码中定义了两个对比函数, 其中 desc 进行降序排列, 但字母不区分大小写; desc1 进行严格降序排列。运行代码, 效果如图 16.37 所示。

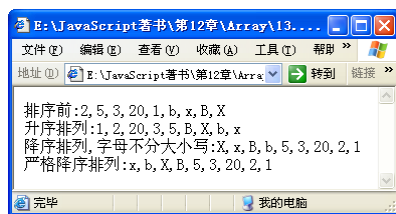


图 16.37 对数组中的元素进行排序

16.5.11 返回一个包含数组中全部数据的字符串

在 JavaScript 中, toString 方法是将数组中的所有元素连接为一个字符串, 各个元素之间使用逗号 (“,”) 连接。其语法如下是。

```
array.toString()
```

下面的代码演示了 toString 方法的使用。

```

<script>
    var arr=new Array(1,2,3,4,"this ia ", "a test");
    document.write(arr.toString());
</script>

```

运行代码，效果如图 16.38 所示。

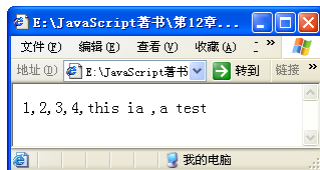


图 16.38 返回一个包含数组中全部数据的字符串

第 17 章 常见 JavaScript 正则表达式应用

JavaScript 提供了丰富的正则表达式功能，其功能涵盖最初的正则表达式建立，以及后期的正则表达式结果处理。本章将重点讲解在 JavaScript 中如何使用正则表达式，并对正则表达式处理结果进行各项处理的问题。

17.1 正则表达式对象 RegExp 及其应用

JavaScript 中使用 RegExp 对象来表述一个正则表达式。本节将详细讲解在 JavaScript 中，如何创建正则表达式对象，并进行匹配检验、编译以及替换。

17.1.1 正则表达式的创建

在 JavaScript 中，通过 RegExp 对象来表述和使用正则表达式。在使用正则表达式之前，首先需要创建一个 RegExp 对象。显示创建 RegExp 对象的语法如下：

```
var oRexp=new RegExp(pattern[,flag])
```

其中，oRexp 为必选项，表示所创建的 RegExp 对象的名称。pattern 为必选项，表示以字符串格式表示的正则表达式。flag 为可选项，是匹配选项，可用的值有四个：g 表示全局匹配检测；i 表示在匹配检测时忽略大小写；gi 表示全局匹配检测且忽略大小写；m 表示允许多行搜索。

用户不一定要显式地创建。另外，还可以隐式创建 RegExp 对象，格式如下：

```
var oRexp=/pattern/[flag]
```

其中，oRexp 为必选项，表示所创建的 RegExp 对象的名称。pattern 为必选项，表示以字符串格式表示的正则表达式。flag 为可选项，匹配选项。

注意：在使用显式格式创建 RegExp 对象时，正则表达式中的“\”要用“\\”来代替，且表达式开头和结尾的“/”需要省略。

17.1.2 判断字符串中是否存在匹配内容

在 JavaScript 中，test 方法用于判断用户输入的数据是否符合指定的规则，如判断 E-mail 地址是否合法等。其返回值为布尔值。通过该值，可以判断字符串中是否存在与正则表达式相匹配的结果。如果返回值为 true，则说明字符串中存在匹配内容；如果为 false，则不存在匹配内容。其语法如下：

```
objReg.test(objStr)
```

其中，objReg 为必选项，表示 RegExp 对象的名称。string 为必选项，表示要进行匹配检测的字符串。下面的代码演示了如何利用 test 方法判断用户输入的 E-mail 地址是否合法。

```
<title>Email 合法性检测</title>
<body>
<script>
function checkEmail()
{
```

```

//创建检测 Email 地址的正则表达式
var objReg=/\w+[@]{1}\w+[\.]{1}\w+/;
var email=document.forms[0].email.value;
if (objReg.test(email))
{
    alert("输入 E-mail 地址符合要求");
}
else
{
    alert("输入 E-mail 地址不合法");
}
}
</script>
<form name="Email">
    请输入 Email 地址:
    <input name="email" type="text">
    <input name="check" type="button" value="检测" onclick="checkEmail()">
</form>
</body>

```

运行上述代码，在文本框中输入 E-mail 地址，然后单击“检测”按钮，即可判断输入的 E-mail 地址是否合法。效果如图 17.1 所示。

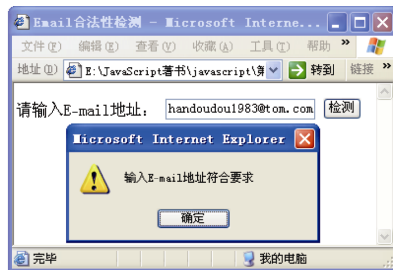


图 17.1 判断输入的 E-mail 地址是否合法

17.1.3 对字符串进行匹配检测

在 JavaScript 中，`exec` 方法通过对指定的字符串进行一次匹配检测，获取字符串中第一个与正则表达式相匹配的内容，并将该匹配内容及其子匹配的结果存储到返回的数组中。其语法如下：`objReg.exec(string)`

其中，`objReg` 为必选项，表示 `RegExp` 对象的名称。`string` 为必选项，表示要进行匹配检测的字符串。下面的代码演示了 `exec` 方法的使用。

```

<script>
    var ostr="我的 13511111111 他的 13222222222 她的 13605461234";
    var reg=/13(\d)(\d{8})/g;
    var arr=reg.exec(ostr);
    document.write("检测到如下手机号: <br>");
    writeNumber(arr);
    //定义函数，输出数组中的内容
    function writeNumber(arr)
    {
        for(var i=0;i<arr.length;i++)
        {
            document.write("<li>" + arr[i]);
        }
    }
}

```

```
</script>
```

运行上述代码，效果如图 17.2 所示。

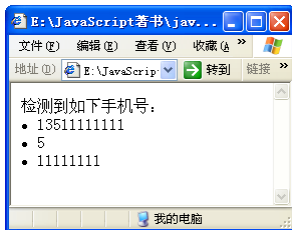


图 17.2 对字符串进行匹配检测

17.1.4 编译正则表达式

在 JavaScript 中，`compile` 方法可以编译指定的正则表达式。编译之后的正则表达式执行速度将会提高。如果该正则表达式要被多次调用，那么调用 `compile` 方法可以有效地提高代码的执行速度。如果该正则表达式只被使用一次，则不会有明显的效果。其语法如下：

```
objReg.compile(pattern[, flag])
```

其中，`objReg` 为必选项，表示 `RegExp` 对象变量的名称。`pattern` 为必选项，表示正则表达式。`flag` 为可选项，表示匹配选项。下面的代码演示了 `compile` 方法的使用。

```
<script>
var ostr="我的 13511111111,他的 132222222222,她的 13605461234";
with(document)
{
    var reg=new RegExp("13[4-9](\\d){8}", "g");
    write("发现移动手机号:");
    writeNumber(ostr.match(reg));
    //重新编译正则表达式
    reg.compile("13[0-3](\\d){8}", "g");
    write("<br>发现联通手机号:");
    writeNumber(ostr.match(reg));
}
//定义函数，输出数组中的内容
function writeNumber(arr)
{
    for(var i=0;i<arr.length;i++)
    {
        document.write("<li>"+arr[i]);
    }
}
</script>
```

上述代码中，首先定义了一个匹配移动手机号的正则表达式。对字符串进行匹配检测后，又定义了一个匹配联通手机号的正则表达式，并进行了匹配检测和输出。运行代码，效果如图 17.3 所示。

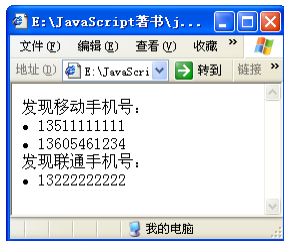


图 17.3 编译正则表达式

17.1.5 替换字符串中的指定内容

在 JavaScript 中，`replace` 方法可以用指定的子字符串替换掉字符串中与指定的正则表达式相匹配的内容。其语法如下：

```
objStr.replace(regex,subStr)
```

其中，`objStr` 为必选项，表示字符串变量或常量。`regex` 为必选项，表示 `RegExp` 对象的名称或者以字符串格式表示的正则表达式。`subStr` 为必选项，表示任意有效的字符串表达式。下面的代码演示了 `replace` 方法的使用。

```
<script>
var ostr=new String("this is a test");
var reG=/s\\w?/g;
var re=/s\\w?/;
with (document)
{
    write(ostr+"<br>");
    //全局检测
    write(ostr.replace(reG,"**"));
    write("<br>");
    //只进行一次匹配渐次
    write(ostr.replace(re,"x"));
}
</script>
```

上述代码中定义了两个 `RegExp` 对象。两个对象所表述的正则表达式相同。但第一个使用的是全局检测匹配，第二个只进行一次匹配检测。运行代码，效果如图 17.4 所示。

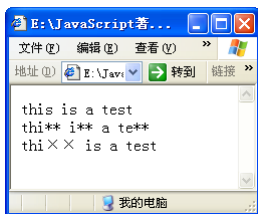


图 17.4 替换字符串中的指定内容

17.2 处理匹配的结果

在 JavaScript 中，使用正则表达式进行检验后，匹配结果将保存在集合中。本节将讲解集合中的各种处理，如获取匹配位置、获取子匹配结果和获取匹配索引。

17.2.1 获取字符串中所有的匹配信息

在 JavaScript 中，如果字符串中的内容与指定的正则表达式至少有一个匹配时，`match` 方法将返回一个数组。数组中存储了字符串中所有的匹配信息。如果没有任何匹配信息，则该方法的返回值为 `null`。其语法如下：

```
objStr.math(regex)
```

其中，`objStr` 为必选项，表示字符串变量或常量。`regex` 为必选项，表示 `RegExp` 对象的名称或者以字符串格式表示的正则表达式。下面的代码演示了如何利用 `math` 方法得到字符串中的全部匹配。

```
<script>
```



```

var ostr="Do you love javascript?";
var re=/\wo(\w+)?/g;
/*该正则表达式匹配所有以一个字符（字母、数字或下划线）开头，
第二个字符为 o，后面有一个或多个字符相随*/
var arr=ostr.match(re);
document.write(ostr);
if (arr!=null)
{
    for(var i=0;i<arr.length;i++)
    {
        document.write("<li>"+arr[i]);
    }
}
</script>

```

上述代码首先建立了一个正则表达式“`^\wo(\w+)?/g`”。其中，“+”表示其前面的表达式要匹配一次或多次。然后，利用 `match` 方法得到了字符串中所有的匹配内容并存储到一个数组中，最后输出了该数组中的全部内容。运行代码，效果如图 17.5 所示。

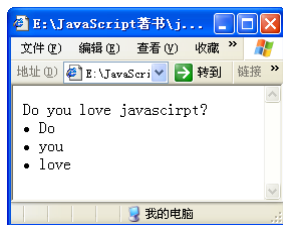


图 17.5 获取字符串中所有的匹配信息

17.2.2 获取第一次匹配的起始位置（1）

在 JavaScript 中，`search` 方法是与指定的正则表达式相比较，以得到与之相匹配的内容第一次出现时的位置。其语法如下：

```
objstr.search(regexp)
```

其中，`objStr` 为必选项，表示字符串变量或常量。`regexp` 为必选项，表示 `RegExp` 对象的名称或者以字符串格式表示的正则表达式。下面的代码演示了如何利用 `search` 方法得到第一次匹配的位置。

```

<script>
var re=/(\d)(\d)\d\2\1/
var ostr="11010111"
var pos=ostr.search(re);
if(pos!=-1)
    document.write("没有找到任何匹配！");
else
{
    arr=ostr.match(re);
    document.write("在"+pos+"找到第一个匹配，匹配内容为：");
    document.write(arr[0]);
}
</script>

```

上述代码定义了一个正则表达式。其中，使用了子匹配的反向引用，用于查找类似于“`mnlm`”的匹配。其中，`m`、`n` 和 `l` 均为单个数字。运行代码，效果如图 17.6 所示。

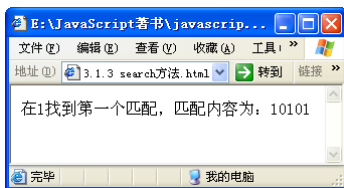


图 17.6 获取第一次匹配的起始位置 (1)

17.2.3 获取第一次匹配的起始位置 (2)

在 JavaScript 中, `index` 属性的值与 `search` 方法的返回值相同, 都是字符串中第一次匹配出现的起始位置, 该位置的计算从 0 开始。如果没有找到任何匹配, 则该属性的值为 -1。其语法如下:

`RegExp.index`

该属性也是 `RegExp` 对象的静态属性, 调用方式固定。下面的代码对比演示了 `index` 属性和 `search` 方法。

```
<script>
  var re=/(\d)(\d)\d\2\1/
  var ostr="1101011100110101100110011001"
  var pos=ostr.search(re);
  if(pos==-1)
    document.write("没有找到任何匹配!");
  else
  {
    arr=ostr.match(re);
    document.write("在"+pos+"找到第一个匹配, 匹配内容为: ");
    document.write(arr[0]);
    document.write("<br>此时 RegExp.index 的值为: "+RegExp.index);
  }
</script>
```

上述代码在完成匹配检测后, 分别输出了 `search` 方法和 `index` 属性的值。运行代码, 效果如图 17.7 所示。

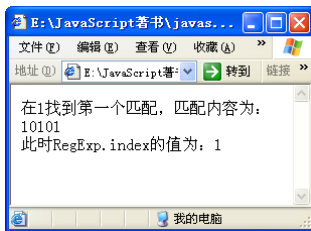


图 17.7 获取第一次匹配的起始位置 (2)

17.2.4 获取子匹配的结果

在 JavaScript 中, `$1~$9` 中存储了正则表达式最近的 9 个子匹配结果。这些匹配结果按子匹配出现的顺序从左到右依次存储。其语法如下:

`[RegExp.]$n`

这些属性为静态的。除了在 `replace` 方法的第二个参数中, 可以省略 `RegExp` 而直接使用外, 在其他地方使用这些属性时, 必须使用 `RegExp.$n` 的调用格式。下面的代码演示了这些属性的使用。

```
<script>
```

```

var re=/ (13) (\d) (\d{8}) /;
var ostr="这是我的手机号 13511111111";
with (document)
{
    write("检测到手机号:<br>" + ostr.replace(re, "$1$2*****"));
    write("<br>");
    if (parseInt(RegExp.$2) < 4)
        write("这是一个联通手机号");
    else
        write("这是一个移动手机号");
}
</script>

```

上述代码定义了一个用于检测手机号的正则表达式。该表达式中利用三个子匹配将该表达式分为三部分。利用第二个子匹配可以判断手机号所属的通信服务商。执行代码，效果如图 17.8 所示。

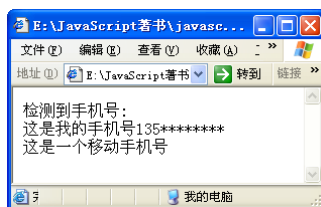


图 17.8 获取子匹配的结果

每进行一次正则表达式的匹配检测，\$1~\$9 的值就会被更新一次。因此这段代码只能提取到一个手机号的信息。实际上在调用 `match` 或 `exec` 等方法时，正则表达式的整体匹配结果会被存放到数组中。因此，可以把上面的代码修改为：

```

<script>
var re=/ (13) (\d) (\d{8}) /g;
var ostr="这是我的手机号 13511111111,他的时 132222222222";
var arr=ostr.match(re);
with (document)
{
    write("检测到"+arr.length+"个手机号:<br>");
    for(var i=0;i<arr.length;i++)
    {
        //对每一个手机号进行新的匹配检测，以更新$1-$9 的值
        arr[i].toString().match(re);
        write("<li>" + arr[i]);
        if (parseInt(RegExp.$2) < 4)
            write("这是一个联通手机号");
        else
            write("这是一个移动手机号");
    }
}
</script>

```

上述代码首先利用 `match` 方法将正则表达式所匹配到的手机号存储到一个数组中，然后利用 `for` 循环，在数组中的每一项上调用 `match` 方法更新 `$n` 的值，以利用这些属性来获得一些手机号的信息。运行代码，效果如图 17.9 所示。

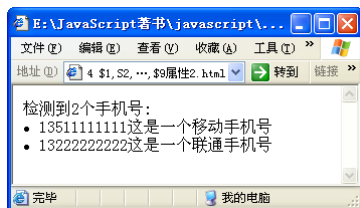


图 17.9 获取子匹配的结果

17.2.5 获取与正则表达式进行匹配检测的字符串

在 JavaScript 中, `input` 属性为 `RegExp` 的静态只读属性, 该属性的值为与 `RegExp` 对象所表述的正则表达式进行匹配检测的字符串。该属性还可以表示为 “\$_”, 其语法如下:

```
RegExp.input
RegExp.$_
```

下面的代码演示了 `input` 属性的使用。

```
<script>
var ostr="abcDdefCDDE";
var re=/cd+e/i;
document.write(ostr+"<br>");
ostr.match(re);
document.write(RegExp.input+"<br>");
document.write(RegExp.$_+"<br>");
</script>
```

运行代码, 效果如图 17.10 所示。从执行结果中可以看出, `input` 与 `$_` 的返回值是相同的。



图 17.10 获取与正则表达式进行匹配检测的字符串

17.2.6 获取最近一次匹配的内容

在 JavaScript 中, `lastMatch` 属性 (`&&`) 也是 `RegExp` 对象的静态只读属性。该属性的值为正则表达式与字符串最后一次匹配的结果。进行任意新的匹配都会改变 `lastMatch` 属性的值。如果没有任何匹配信息, 则该属性的值为空字符串。其语法如下:

```
RegExp.lastIndex
```

下面的代码演示了 `lastMatch` 属性 (`&&`) 的使用。

```
<script>
var ostr="abcDdefCDDE";
var re=/cd+e/i;
ostr.match(re);
with(document)
{
    write(ostr);
    write("<br>最后的匹配为: "+RegExp.lastMatch);
    "abcdef".match(re);
    write("<br>最后的匹配为: "+RegExp.lastMatch);
}
```

```
</script>
```

运行代码，效果如图 17.11 所示。

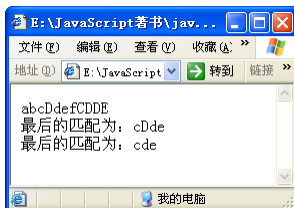


图 17.11 获取最近一次匹配的内容

17.2.7 获取最近一次匹配的最后个子匹配

在 JavaScript 中，`lastParen` 属性（\$+）为 `RegExp` 属性的只读静态属性。该属性返回在最近一次匹配检测中所得到的最后一个子匹配的值。任何新的匹配检测都会改变该属性的值。如果没有任何子匹配，则该属性的值为一个空字符串。其语法如下：

```
RegExp.lastParen
```

下面的代码演示了 `lastParen` 属性（\$+）的使用。

```
<script>
var ostr="abcDdefCDDE";
var re=/c(d+)e/i;
ostr.match(re);
with(document)
{
    write(ostr);
    write("<br>最后的子匹配为: "+RegExp.lastParen);
    "abcdef".match(re);
    write("<br>最后的子匹配为: "+RegExp.lastParen);
}
</script>
```

运行代码，效果如图 17.12 所示。

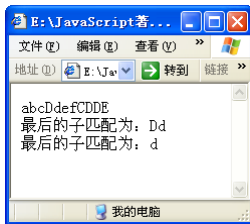


图 17.12 获取最近一次匹配的最后个子匹配

17.2.8 获取匹配的内容的最后一个索引位置

在 JavaScript 中，`lastIndex` 属性为 `RegExp` 对象的静态属性。其值为字符串中的匹配内容在字符串中的最后一个匹配位置，也就是进行下一次匹配查找的起始位置。其语法如下：

```
RegExp.lastIndex
```

下面的代码演示了 `lastIndex` 属性的使用。

```
<script>
var ostr="abcDdefCDDE";
var re=/cd+e/i;
with(document)
```

```

{
    write(ostr+"<br>");
    write("发现匹配字符串"+ostr.match(re));
    write("<br>起始位置:"+RegExp.index);
    write("<br>结束位置: "+RegExp.lastIndex);
}
</script>

```

上述代码分别输出了匹配内容、该内容的起始位置和结束位置。运行代码，效果如图 17.13 所示。

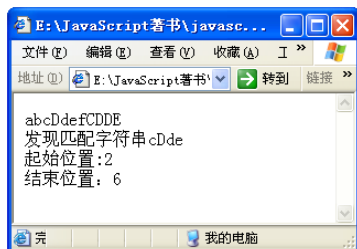


图 17.13 获取匹配的内容的最后一个索引位置

17.2.9 获取匹配内容左侧的字符信息

在 JavaScript 中，`leftContext` 属性为 `RegExp` 对象的只读静态属性。该属性的值为从字符串开始到最后一次匹配之前的字符串。该属性的初始值为空字符串，如果检测到相应的匹配，则该属性的值就会改变。其语法如下：

```
RegExp.leftContext
```

以下代码演示使用 `leftContext` 获取匹配之前的内容。

```

<script>
    var ostr="abcDdefCDDE";
    var re=/c(d+)e/i;
    ostr.match(re);
    with(document)
    {
        write(ostr);
        write("<br>最后的子匹配为: "+RegExp.lastParen);
        write("<br>在此之前的内容为: "+RegExp.leftContext);
        ostr="opqrst";
        write("<br>"+ostr);
        write("<br>在此之前的内容为: "+RegExp.leftContext);
    }
</script>

```

运行代码，效果如图 17.14 所示。

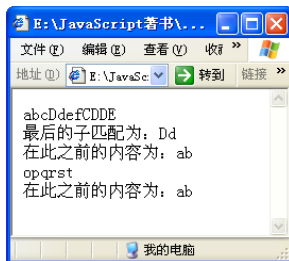


图 17.14 获取匹配内容左侧的字符信息

17.2.10 获取匹配内容右侧的字符信息

在 JavaScript 中, `rightContext` 属性 (`$'`) 为 `RegExp` 对象的只读静态属性。该属性的值为从最后一次匹配之后到字符串末尾的字符串。该属性的初始值为空字符串。如果检测到相应的匹配, 则该属性的值就会改变。其语法如下:

`RegExp.rightContext`

下面的代码演示了 `rightContext` 属性 (`$'`) 的使用。

```
<script>
  var ostr="abcDdef";
  var re=/bcd+e/i;
  ostr.match(re);
  with(document)
  {
    write("原始字符串为: "+ostr);
    write("<br>匹配内容为: "+ostr.match(re));
    write("<br>在此之前的内容为: "+RegExp.leftContext);
    write("<br>在此之后的内容为: "+RegExp.leftContext);
    write("<br>它们的组合为: "+RegExp.leftContext+ostr.match(re)+RegExp.rightContext);
  }
</script>
```

上述代码分别输出了原始字符串、匹配内容、匹配内容的左侧内容和右侧内容以及它们的组合。运行代码, 效果如图 17.15 所示。

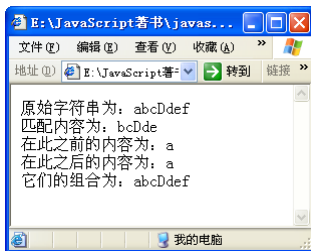


图 17.15 获取匹配内容右侧的字符信息